Code Complete

Steve McConnell

8. Defensive Programming

DEFENSIVE PROGRAMMING DOESN'T MEAN being defensive about your programming—"It does so work!" The idea is based on defensive driving. In defensive driving, you adopt the mind-set 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.

In defensive programming, the main idea is that if a routine is passed bad data, it won't be hurt, even if the bad data is another routine's fault. More generally, it's the recognition that programs will have problems and modifications, and that a smart programmer will develop code accordingly.

Presentation 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

In school you might have heard the expression, “Garbage in, garbage out.” That expression is essentially software development’s version of caveat emptor: let the user beware.

For production software, garbage in, garbage out isn’t good enough. A good program never puts out garbage, regardless of what it takes in. A good program uses “garbage in, nothing out”; “garbage in, error message out”; or “no garbage allowed in” instead. By today’s standards, “garbage in, garbage out” is the mark of a sloppy, nonsecure program.

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. Make sure that numeric values are within tolerances and that strings are short enough to handle. If a string is intended to represent a restricted range of values (such as a financial transaction ID or something similar), be sure that the string is valid for its intended purpose; otherwise reject it.

8.1. Protecting Your Program From Invalid Inputs

If you're working on a secure application, be especially leery of data that might attack your system: attempted buffer overflows, injected SQL commands, injected html or XML code, integer overflows, and so on.

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.

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 later.

8.1. Protecting Your Program From Invalid Inputs

Defensive programming is useful as an adjunct to the other techniques for quality improvement described in this book. The best form of defensive coding is not inserting errors in the first place. Using iterative design, writing pseudocode before 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. Fortunately, you can use defensive programming in combination with the other techniques.

8.1. Protecting Your Program From Invalid Inputs

As Figure suggests, protecting yourself from seemingly small problems can make more of a difference than you might think. The rest of presentation describes specific options for checking data from external sources, checking input parameters, and handling bad inputs..

Figure 8-1
Part of the Interstate-90 floating bridge in Seattle sank during a storm because the flotation tanks were left uncovered, they filled with water, and the bridge became too heavy to float. During construction, protecting yourself against the small stuff matters more than you might think.

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. For example, if the system assumes that a customer-information file will never have more than 50,000 records, the program might contain an assertion that the number of records is less than or equal to 50,000. As long as the number of records is less than or equal to 50,000, the assertion will be silent. If it encounters more than 50,000 records, however, it will loudly "assert" that there is an error in the program.

Assertions are especially useful in large, complicated programs and in high reliability programs. They enable programmers to more quickly flush out mismatched interface assumptions, errors that creep in when code is modified, and so on.

8.2. Assertions

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.

Here's what a Java assertion would look like if the variable denominator were expected to be nonzero:

```
assert denominator != 0 : "denominator is unexpectedly equal to 0.";
```

This assertion asserts that denominator is not equal to 0. The first argument, *denominator != 0*, is a boolean expression that evaluates to True or False. The second argument is a message to print if the first argument is False — that is, if the assertion is false.

8.2. Assertions

Use assertions to document assumptions made in the code and to flush out unexpected conditions. Assertions can be used to check assumptions like these:

- That an input parameter's value falls within its expected range (or an output parameter's value does)
- That a file or stream is open (or closed) when a routine begins executing (or when it ends executing)
- That a file or stream is at the beginning (or end) when a routine begins executing (or when it ends executing)
- That a file or stream is open for read-only, write-only, or both read and write
- That the value of an input-only variable is not changed by a routine
- That a pointer is non-NULL
- That an array or other container passed into a routine can contain at least X number of data elements
- That a table has been initialized to contain real values
- That a container is empty (or full) when a routine begins executing (or when it finishes)

8.2. Assertions

● That the results from a highly optimized, complicated routine match the results from a slower but clearly written routine
● Etc.

Of course, these are just the basics, and your own routines will contain many more specific assumptions that you can document using assertions.

Normally, you don’t want users to see assertion messages in production code; assertions are primarily for use during development and maintenance. Assertions are normally compiled into the code at development time and compiled out of the code for production. During development, assertions flush out contradictory assumptions, unexpected conditions, bad values passed to routines, and so on. During production, they are compiled out of the code so that the assertions don’t degrade system performance.

8.2. Assertions

Guidelines for Using Assertions

Here are some 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 code checks for off-nominal circumstances that might not occur very often, but that have been anticipated by the programmer who wrote the code and that need to be handled by the production code. Error-handling typically checks for bad input data; assertions check for bugs in the code.

If error handling code is used to address an anomalous condition, the error handling will enable the program to respond to the error gracefully. 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.

8.2. Assertions

Avoid putting executable code in assertions

Putting code into an assertion raises the possibility that the compiler will eliminate the code when you turn off the assertions.

The problem with this code is that, if you don’t compile the assertions, you don’t compile the code that performs the action. Put executable statements on their own lines, assign the results to status variables, and test the status variables instead.

8.2. Assertions

Use assertions to document 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. Preconditions are the client code’s obligations to the code it calls.

Postconditions are the properties that the routine or class promises will be true when it concludes executing. Postconditions are the routine or class’s obligations to the code that uses it.

Assertions are a useful tool for documenting preconditions and postconditions. Comments could be used to document preconditions and postconditions, but, unlike comments, assertions can check dynamically whether the preconditions and postconditions are true.

8.2. Assertions

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.

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 at different points in the system's lifetime.

On a large development team, some programmers will inevitably be more conscientious than others and some parts of the code will be reviewed more rigorously than other parts of the code.

8.2. Assertions

With test teams working across different geographic regions and subject to business pressures that result in test coverage that varies with each release, you can’t count on comprehensive regression testing, either.

In such circumstances, both assertions and error handling code might be used to address the same error. In the source code for Microsoft Word, for example, conditions that should always be true are asserted, but such errors are also handled by error-handling code in case the assertion fails.

For extremely large, complex, long-lived applications like Word, assertions are valuable because they help to flush out as many development-time errors as possible. But the application is so complex (million of lines of code) and has gone through so many generations of modification that it isn’t realistic to assume that every conceivable error will be detected and corrected before the software ships, and so errors must be handled in the production version of the system as well.

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?

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 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.

8.3. Error Handling Techniques

Return the same answer as the previous time

If the thermometer-reading software doesn't get a reading one time, it might simply return the same value as last time. Depending on the application, temperatures might not be very likely to change much in 1/100th of a second. In a video game, 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. This is often a reasonable approach when taking readings from a calibrated instrument. The thermometer might be calibrated between 0 and 100 degrees Celsius, for example. If you detect a reading less than 0, you can substitute 0 which is the closest legal value. If you detect a value greater than 100, you can substitute 100. For a string operation, if a string length is reported to be less than 0, you could substitute 0. My car uses this approach to error handling whenever I back up. Since my speedometer doesn't show negative speeds, when I back up it simply shows a speed of 0—the closest legal value.

8.3. Error Handling Techniques

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. The specific mechanism for notifying the rest of the system that an error has occurred could be any of the following:

- Set the value of a status variable
- Return status as the function's return value
- Throw an exception using the language's built-in exception mechanism

In this case, the specific error-reporting mechanism is less important than the decision about which parts of the system will handle errors directly and which will just report that they've occurred. If security is an issue, be sure that calling routines always check return codes***.***

8.3. Error Handling Techniques

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, try to clearly separate the UI from the rest of the system, or try to localize the software into a different language. Also, beware of telling a potential attacker of the system too much. Attackers sometimes use error messages to discover how to attack a system.

8.3. Error Handling Techniques

Handle the error in whatever way works best locally

Some designs call for handling all errors locally—the decision of which specific error-handling method to use is left up to the programmer designing and implementing the part of the system that encounters the error.

This approach provides individual developers with great flexibility, but it creates a significant risk that the overall performance of the system will not satisfy its requirements for correctness or robustness (more on this later). Depending on how developers end up handling specific errors, this approach also has the potential to spread user interface code throughout the system, which exposes the program to all the problems associated with displaying error messages.

8.3. Error Handling Techniques

Shutdown

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

For example, 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. We'd much prefer to reboot the machine than to run the risk of delivering the wrong dosage.

A similar approach can be used to improve security of Microsoft Windows. By default, Windows continues to operate even when its security log is full. But you can configure Windows to halt the server if the security log becomes full, which can be appropriate in a security-critical environment.

8.3. Error Handling Techniques

Robustness vs. Correctness

Here's a brain teaser:

Suppose an application displays graphic information on a screen. An error condition results in a few pixels in the lower right quadrant displaying in the wrong color. On next update, the screen will refresh, and the pixels will be the right color again. What is the best error processing approach?

What do you think is the best approach? Is it to use the same value as last time?

Or perhaps to use the closest legal value? Suppose this error occurs inside a fast paced video game, and the next time the screen is refreshed the pixels will be repainted to be the right color (which will occur within less than one second)? In that case, choose an approach like using the same color as last time or using the default background color.

8.3. Error Handling Techniques

Now suppose that the application is not a video game, but software that displays X-rays. Would using the same color as last time be a good approach, or using the default background color? Developers of that application would not want to run the risk of having bad data on an X-ray, and so displaying an error message or shutting down would be better ways to handle that kind of error.

The style of error processing that is most appropriate depends on the kind of software the error occurs in and generally favors more correctness or more robustness. Developers tend to use these terms informally, but, strictly speaking, these terms are at opposite ends of the scale from each other. ***Correctness*** means never returning an inaccurate result; no result is better than 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.

8.3. Error Handling Techniques

Safety critical applications tend to favor correctness to robustness. It is better to return no result than to return a wrong result. The radiation machine is a good example of this principle.

Consumer applications tend to favor robustness to correctness. Any result whatsoever is usually better than the software shutting down. The word processor I’m using occasionally displays a fraction of a line of text at the bottom of the screen. If it detects that condition do I want the word processor to shut down? No. I know that the next time I hit page up or page down, the screen will refresh, and the display will be back to normal.

8.3. Error Handling Techniques

High-Level Design Implications of Error Processing

Deciding on a general approach to bad parameters is an architectural or high-level design decision and should be addressed at one of those levels.

Once you decide on the approach, make sure you follow it consistently. If you decide to have high-level code handle errors and low-level code merely report errors, make sure the high level code actually handles the errors! Some languages including C++ might give you the option of ignoring the fact that a function is returning an error code. Don't ignore error information! Test the function return value. If you don't expect the function ever to produce an error, check it anyway. The whole point of defensive programming is guarding against errors you don't expect.

This guideline holds true for system functions as well as your own functions. Unless you've set an architectural guideline of not checking system calls for errors, check for error codes after each call. If you detect an error, include the error number and the description of the error.

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!" Code that has no sense of the context of an error can return control to other parts of the system that might have a better ability to interpret the error and do something useful about it.

The basic structure of an exception in C++, Java, Visual Basic and C# is that a routine uses throw to throw an exception object. Code in some other routine up the calling hierarchy will catch the exception within a try-catch block.

8.4. Exceptions

Table 8-1. Popular Language Support for Exceptions

Exception Attribute	C++	Java	Visual Basic
Try-catch support	yes	yes	yes
Try-catch-finally support	no	yes	yes
What can be thrown	*Exception* object or object derived from *Exception* class; object pointer; object reference; data type like string or int	*Exception* object or object derived from *Exception* class	*Exception* object or object derived from *Exception* class

8.4. Exceptions

Exception Attribute	C++	Java	Visual Basic
Effect of uncaught exception	Invokes *std::unexpected()*, which by default invokes *std::terminate()*, which by default invokes *abort()*	Terminates thread of execution	Terminates program
Exceptions thrown must be defined in class interface	No	Yes	No
Exceptions caught must be defined in class interface	No	Yes	No

8.4. Exceptions

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

Suggestions for realizing the benefits of exceptions and avoiding the difficulties often associated with them are:

Use exceptions to notify other parts of the program about errors that should not be ignored

The overriding benefit of exceptions is their ability to signal error conditions in such a way that they cannot be ignored (Meyers 1996).

Other approaches to handling errors create the possibility that an error condition can propagate through a code base undetected. Exceptions eliminate that possibility.

8.4. Exceptions

Throw an exception only for conditions that are truly exceptional

Exceptions should be reserved for conditions that are truly exceptional, in other words, conditions that cannot be addressed by other coding practices. Exceptions are used in similar circumstances to assertions—for events that are not just infrequent, but 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. That increases code complexity, which works against so called Software's Major Technical Imperative: Managing Complexity.

Don't use an exception to pass the buck

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.

8.4. Exceptions

Avoid throwing exceptions in constructors and destructors unless you catch them in the same place

The rules for how exceptions are processed become very complicated very quickly when exceptions are thrown in constructors and destructors. In C++, for example, destructors aren't called unless an object is fully constructed, which means if code within a constructor throws an exception, the destructor won't be called, and that sets up a possible resource leak (Meyers 1996, Stroustrup 1997).

Similarly complicated rules apply to exceptions within destructors. Language lawyers might say that remembering rule like these is "trivial," but programmers who are mere mortals will have trouble remembering them. It's better programming practice simply to avoid the extra complexity such code creates by not writing that kind of code in the first place.

Throw exceptions at the right level of abstraction

A routine should present a consistent abstraction in its interface, and so should a class. The exceptions thrown are part of the routine interface, just like specific data types are.

8.4. Exceptions

When you choose to pass an exception to the caller, make sure the exception's level of abstraction is consistent with the routine interface's abstraction. Here is an example of what not to do:

```
class Employee {
   ...
   public TaxId getTaxId() EOFException {
      ...
   }
   ...
}
```

The getTaxId() code passes the lower-level io_disk_not_ready exception back to its caller. It doesn't take ownership of the exception itself; it exposes some details about how it is implemented by passing the lower-level exception to its caller. This effectively couples the routine's client's code not the Employee class's code, but to the code below the Employee class that throws the io_disk_not_ready exception. Encapsulation is broken, and intellectual manageability starts to decline.

8.4. Exceptions

Instead, the getTaxId() code should pass back an exception that's consistent with the class interface of which it's a part, like this:

```
class Employee {
   ...
   public TaxId getTaxId() throws EmployeeDataNotAvailable {
      ...
   }
   ...
}
```

The exception-handling code inside getTaxId() will probably just map the io_disk_not_ready exception onto the EmployeeDataNotAvailable exception, which is fine because that's sufficient to preserve the interface abstraction.

8.4. Exceptions

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

Every exception occurs in specific circumstances that are detected at the time the code throws the exception. This information is invaluable to the person who reads the exception message. Be sure the message contains the information needed to understand why the exception was thrown. If the exception was thrown because of an array index error, be sure the exception message includes the upper and lower array limits and the value of the illegal index.

Avoid empty catch blocks

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

```
try {
   ...
   // lots of code
  ...
} catch ( AnException exception ) {
}
```

8.4. Exceptions

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. The centralized exception reporter provides a central repository for knowledge about what kinds of exceptions there are, how each exception should be handled, formatting of exception messages, and so on.

8.4. Exceptions

Standardize your project's use of exceptions

To keep exception handling as intellectually manageable as possible, you can standardize your use of exceptions in several ways.

- If you're working in a language like C++ that allows you to throw a variety of kinds of objects, data, and pointers, standardize on what specifically you will throw. For compatibility with other languages, consider throwing only objects derived from the Exception base class.
- Define the specific circumstances under which code is allowed to use throw catch syntax to perform error processing locally.
- Define the specific circumstances under which code is allowed to throw an exception that won't be handled locally.
- Determine whether a centralized exception reporter will be used.
- Define whether exceptions are allowed in constructors and destructors.

8.4. Exceptions

Consider alternatives to exceptions

Several programming languages have supported exceptions for 5-10 years or more, but little conventional wisdom has emerged about how to use them safely.

Some programmers use exceptions to handle errors just because their language provides that particular error-handling mechanism. You should always consider the full set of error-handling alternatives: handling the error locally, propagating the error using an error code, logging debug information to a file, shutting down the system, or using some other approach.

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).

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. They are also similar to firewalls in a building. A building's firewalls prevent fire from spreading from one part of a building to another part. Barricades used to be called "firewalls," but the term "firewall" now commonly refers to port blocking.

One way to barricade for defensive programming purposes is to designate certain interfaces as boundaries to "safe" areas. Check data crossing the boundaries of a safe area for validity and respond sensibly if the data isn't valid.

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.

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

Another way of thinking about this approach is as an operating-room technique. Data is sterilized before it's allowed to enter the operating room. Anything that's in the operating room is assumed to be safe. The key design decision is deciding what to put in the operating room, what to keep out, and where to put the doors—which routines are considered to be inside the safety zone, which are outside, and which sanitize the data. The easiest way to do this is usually by sanitizing external data as it arrives, but data often needs to be sanitized at more than one level, so multiple levels of sterilization are sometimes required.

Convert input data to the proper type at input time

Input typically arrives in the form of a string or number. Sometimes the value will map onto a boolean type like "yes" or "no." Sometimes the value will map onto an enumerated type like Color_Red, Color_Green, and Color_Blue. Carrying data of questionable type for any length of time in a program increases complexity and increases the chance that someone can crash your program by inputting a color like "Yes." Convert input data to the proper form as soon as possible after it's input.

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. The production version shouldn't expose dangerous operations to the user. The development version can have extra operations that you can use without a safety net.

One program I worked on made extensive use of a quadruply linked list. The linked-list code was error prone, and the linked list tended to get corrupted. I added a menu option to check the integrity of the linked list. Be willing to trade speed and resource usage during development in exchange for built-in tools that can make development go more smoothly.

8.6. Debugging Aids

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. If you write the aid after the first time, however, or use one from a previous project, it will help throughout the project.

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" (Howard and LeBlanc 2003).

Suppose you have a case statement that you expect to handle only five kinds of events. During development, the default case should be used to generate a warning that says "Hey! There's another case here! Fix the program!" During production, however, the default case should do something more graceful, like writing a message to an error-log file.

8.6. Debugging Aids

Here are some ways you can program offensively:

● 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.

● Completely fill any memory allocated so that you can detect memory allocation errors.

● Completely fill any files or streams allocated to flush out any file-format errors.

● Be sure the code in each case statement's else clause fails hard (aborts the program) or is otherwise impossible to overlook.

● Fill an object with junk data just before it's deleted

Sometimes the best defense is a good offense. Fail hard during development so that you can fail softer during production.

8.6. Debugging Aids

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 and build tools like make

Version-control tools can build different versions of a program from the same source files. In development mode, you can set the build tool to include all the debug code. In production mode, you can set it to exclude any debug code you don't want in the commercial version.

Use a built-in preprocessor

If your programming environment has a preprocessor—as C++ does, for example—you can include or exclude debug code at the flick of a compiler switch. You can use the preprocessor directly or by writing a macro that works with preprocessor definitions.

8.6. Debugging Aids

Here's an example of writing code using the preprocessor directly:

```
#define DEBUG
...

#if defined( DEBUG )
// debugging code
...

#endif
```

This theme has several variations. Rather than just defining DEBUG, you can assign it a value and then test for the value rather than testing whether it's defined. That way you can differentiate between different levels of debug code.
You might have some debug code that you want in your program all the time, so you surround that by a statement like #if DEBUG > 0. Other debug code might be for specific purposes only, so you can surround it by a statement like #if DEBUG == POINTER_ERROR. In other places, you might want to set debug levels, so you could have statements like #if DEBUG > LEVEL_A.

8.6. Debugging Aids

If you don’t like having #if defined() spread throughout your code, you can write a preprocessor macro to accomplish the same task. Here’s an example:

```
#define DEBUG

#if defined( DEBUG )
#define DebugCode( code_fragment )   { code_fragment }
#else
#define DebugCode( code_fragment )
#endif
...
```

```
DebugCode(
   statement 1;
   statement 2;
   ...
   statement n;
);
...
```

8.6. Debugging Aids

Write your own preprocessor

If a language doesn't include a preprocessor, it's fairly easy to write one for including and excluding debug code. Establish a convention for designating debug code and write your precompiler to follow that convention. For example, in Java you could write a precompiler to respond to the keywords //#BEGIN DEBUG and //#END DEBUG. Write a script to call the preprocessor, and then compile the processed code. You'll save time in the long run, and you won't mistakenly compile the unpreprocessed code.

8.6. Debugging Aids

Use debugging stubs

In many instances, you can call a routine to do debugging checks. During development, the routine might perform several operations before control returns to the caller.

C++ Example of a Routine that Uses a Debugging Stub

```
void DoSomething(
   SOME_TYPE *pointer;
   ...
   ) {

   // check parameters passed in
   CheckPointer( pointer );
   ...

}
```

This approach incurs only a small performance penalty, and it's a quicker solution than writing your own preprocessor. Keep both the development and production versions of the routines so that you can switch back and forth during future development and production.

8.6. Debugging Aids

For production code, you can replace the complicated routine with a stub routine that merely returns control immediately to the caller or performs only a couple of quick operations before returning control.

C++ Example of a Routine for Checking Pointers During Development

```
void CheckPointer( void *pointer ) {
   // perform check 1--maybe check that it's not NULL
   // perform check 2--maybe check that its dogtag is legitimate
   // perform check 3--maybe check that what it points to isn't corrupted
   ...
   // perform check n--...
}
```

C++ Example of a Routine for Checking Pointers During Production

```
void CheckPointer( void *pointer ) {
   // no code; just return to caller
}
```

8.7. 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. Here are some guidelines for deciding which defensive programming tools to leave in your production code and which to leave out:

Leave in code that checks for important errors

Decide which areas of the program can afford to have undetected errors and which areas cannot. For example, if you were writing a spreadsheet program, you could afford to have undetected errors in the screen-update area of the program because the main penalty for an error is only a messy screen. You could not afford to have undetected errors in the calculation engine because the errors might result in subtly incorrect results in someone's spreadsheet. Most users would rather suffer a messy screen than incorrect tax calculations and an audit by the IRS.

8.7. Determining How Much Defensive Programming to Leave in Production Code

Remove code that checks for trivial errors

If an error has truly trivial consequences, remove code that checks for it. In the previous example, you might remove the code that checks the spreadsheet screen update. “Remove” doesn’t mean physically remove the code. It means use version control, precompiler switches, or some other technique to compile the program without that particular code. If space isn’t a problem, you could leave in the error-checking code but have it log messages to an error-log file unobtrusively.

Remove code that results in hard crashes

During development, when your program detects an error, you’d like the error to be as noticeable as possible so that you can fix it. Often, the best way to accomplish such a goal is to have the program print a debugging message and crash when it detects an error.

During production, your users need a chance to save their work before the program crashes and are probably willing to tolerate a few anomalies in exchange for keeping the program going long enough for them to do that. Users don’t appreciate anything that results in the loss of their work, regardless of how much it helps debugging. If your program contains debugging code that could cause a loss of data, take it out of production.

8.7. Determining How Much Defensive Programming to Leave in Production Code

Leave in code that helps the program crash gracefully

The opposite is also true. If your program contains debugging code that detects potentially fatal errors, leave the code in that allows the program to crash gracefully. In the Mars Pathfinder, for example, engineers left some of the debug code in by design. An error occurred after the Pathfinder had landed. By using the debug aids that had been left in, engineers at JPL were able to diagnose the problem and upload revised code to the Pathfinder, and the Pathfinder completed its mission perfectly (March 1999).

Log errors for your technical support personnel

Consider leaving debugging aids in the production code but changing their behavior so that it's appropriate for the production version. If you've loaded your code with assertions that halt the program during development, you might considering changing the assertion routine to log messages to a file during production rather than eliminating them altogether.

8.7. Determining How Much Defensive Programming to Leave in Production Code

See that the error messages you leave in are friendly

If you leave internal error messages in the program, verify that they're in language that's friendly to the user. In one of my early programs, I got a call from a user who reported that she'd gotten a message that read "You've got a bad pointer allocation, Dog Breath!" Fortunately for me, she had a sense of humor. A common and effective approach is to notify the user of an "internal error" and list an email address or phone number the user can use to report it.

8.8. Being Defensive About Defensive Programming

Too much defensive programming creates problems of its own. If you check data passed as parameters in every conceivable way in every conceivable place, your program will be fat and slow. What's worse, the additional code needed for defensive programming adds complexity to the software. Code installed for defensive programming is not immune to defects, and you're just as likely to find a defect in defensive-programming code as in any other code—more likely, if you write the code casually.

Think about where you need to be defensive, and set your defensive-programming priorities accordingly.

CHECKLIST: Defensive Programming

 Does the routine protect itself from bad input data?
 Have you used assertions to document assumptions, including preconditions and postconditions?
 Have assertions been used only to document conditions that should never occur?
 Does the architecture or high-level design specify a specific set of error handling techniques?
 Does the architecture or high-level design specify whether error handling should favor robustness or correctness?
 Have barricades been created to contain the damaging effect of errors and reduce the amount of code that has to be concerned about error processing?
 Have debugging aids been used in the code?
 Has information hiding been used to contain the effects of changes so that they won't affect code outside the routine or class that's changed?
 Have debugging aids been installed in such a way that they can be activated or deactivated without a great deal of fuss?
 Is the amount of defensive programming code appropriate—neither too much nor too little?

CHECKLIST: Defensive Programming

 Have you used offensive programming techniques to make errors difficult to overlook during development?

Exceptions

 Has your project defined a standardized approach to exception handling?
 Have you considered alternatives to using an exception?
 Is the error handled locally rather than throwing a non-local exception if possible?
 Does the code avoid throwing exceptions in constructors and destructors?
 Are all exceptions at the appropriate levels of abstraction for the routines that throw them?
 Does each exception include all relevant exception background information?
 Is the code free of empty catch blocks? (Or if an empty catch block truly is appropriate, is it documented?)

CHECKLIST: Defensive Programming

Security Issues

 Does the code that checks for bad input data check for attempted buffer overflows, SQL injection, html injection, integer overflows, and other malicious inputs?

 Are all error-return codes checked?

 Are all exceptions caught?

 Do error messages avoid providing information that would help an attacker break into the system?