[Maguire 93] Steve Maguire : "Writing solid code". Microsoft Press 1993.

A

CODING CHECKLISTS

To remind you of the most important points in the book, I've created several checklists you can review during the primary development steps: design, implementation, *DEBUG* support, testing, and debugging. I haven't listed the points that have to do with development overall—I assume that you're using your optional compiler warnings, maintaining *DEBUG* versions of your program, fixing bugs as they are reported, and so on.

To make effective use of these checklists, review them each time you add new code to your project. From a practical standpoint, "each time" really means the "next few times" you write new code. After that, you should have developed a sixth sense for code that bends or breaks the guidelines.

DESIGN

When you consider different designs for a feature, don't stop with the design that gives you the fastest or the smallest result. Consider the risks involved in implementing, maintaining, and using the code that will result from your design. For each possible design, review these points.

- Does this design include undefined or meaningless behavior? What about random or rare behavior? Does the design allow unnecessary flexibility or make unnecessary assumptions? Are there arbitrary details in the design?
- Do you pass any data in static or global buffers? Do any functions rely on the internal workings of other functions? Do any functions do more than one task?
- Does your design have to handle any special cases? Have you isolated the code that handles those special cases?
- Look at the inputs and outputs of your functions. Does each of the inputs and outputs represent exactly one type of data, or do some of them contain error values or other hard-to-notice values? Robust interfaces make every input and output explicit so that programmers can't miss important details such as the *NULL* error value returned by *malloc*, or the fact that *realloc* can release a memory block if you pass in a size of *0*.
- Anticipate how programmers will call your functions. Does the "obvious" approach work correctly? Recall that in *realloc*'s case, the obvious approach creates lost memory blocks.
- On the maintenance side, are your functions readable at the point of call? Each function should perform one task, and its arguments should make the meaning of the call clear. The presence of *TRUE* and *FALSE* arguments often indicates that a function is doing more than one task, or that it is not well designed.
- Do any of your functions return error values? Is it possible to redefine those functions to eliminate the error conditions? Remember that when a function returns an error, that error must be handled—or mishandled—at every point of call.

- Most important, is it possible to automatically and thoroughly validate the design using a unit test? If not, you should consider using an alternative design that can be tested.

IMPLEMENTATION

After implementing your design, you should review these points to ensure that your implementation is robust and error resistant.

- Compare your implementation to your design. Have you accurately implemented the design? Be careful. Minor differences between your design and your implementation can trip you up. Remember the *UnsToStr* example that broke because it used non-negative integers when the design called for integers that were unsigned.
- Do you make unnecessary assumptions in the code? Have you used nonportable data types when portable data types would work? Are there any arbitrary aspects of the implementation?
- Examine the expressions in your code. Can any of them overflow or underflow? What about your variables?
- Have you used nested *?:* operators or other risky C language idioms such as shifting to divide? Have you mixed bitwise operators and arithmetic operators without good cause? Have you used any C idioms in a questionable way? For example, using the 0/1 result of a logical expression in an arithmetic context? Rewrite risky expressions using comparable yet safer expressions.
- Take a close look at your code. Have you used any arcane C that the average programmer on your team wouldn't understand? Consider rewriting the code using mainstream C.
- Each of your functions probably does a single task, but is that task implemented using a single code path, or is the task actually achieved using different code to implement various special cases? If the task is implemented using special-case code, can you eliminate those special cases by using an alternative algorithm? Try to eliminate every *if* statement in your code.

- Do you call any functions that return errors? Can you alter your design so that the call is unnecessary and thus eliminate the need to do error handling?
- Do you reference memory you have no right to touch? Specifically, do you reference memory you have released? Do you peek at private data structures owned by other subsystems?
- If your functions take pointers to inputs or to outputs, does your code restrict its references to only the memory required to hold those inputs and outputs? If not, your code may be making an erroneous assumption about how much memory the caller has allocated for that data.

ADDING *DEBUG* SUPPORT

Adding assertions and other debugging code to your implementations can reduce the time required to find any bugs hiding in your code. This checklist points out worthwhile assertions and debugging code you should consider using.

- Have you used assertions to validate your function arguments? If you find that you can't validate a particular argument because you don't have enough information, would maintaining extra debug information help? Recall how the debug-only *sizeofBlock* function was useful in validating pointers to allocated memory.
- Have you used assertions to validate your assumptions, or to detect illegal uses of undefined behavior? Asserting for undefined behavior prevents programmers from abusing unspecified details of your implementations.
- Defensive programming "fixes" internal bugs when they occur, making such bugs hard to spot. Have you used assertions to detect these bugs in the *DEBUG* version of your program? (Of course, this view of defensive programming doesn't apply to defensive programming used to correct bad end-user inputs.)
- Are your assertions clear? If not, be sure to include comments to explain the tests. Unfortunately, when programmers get an assertion failure and don't understand the purpose of the test, they will often assume that the assertion is invalid and remove it. Comments help preserve your assertions.

- If your code allocates memory, have you used debug-only code to set the uninitialized contents to a known but obviously garbage state? Setting memory to a consistent value will make it easier to find and reliably reproduce bugs that use uninitialized memory.
- If your code releases memory, does it first destroy the contents so that you don't have valid-looking garbage hanging around?
- Are any of your algorithms critical enough that you should use a second, but different, debug-only algorithm to verify the primary one?
- Are there any debug checks you can make at program startup to detect bugs at the earliest possible moment? In particular, are there any data tables you could validate at program startup?

TESTING

It is vitally important that programmers test their code, even if it means slipping the schedule. The questions in this section point out the most beneficial testing steps to take.

- Does the code compile without generating any warnings, including all optional compiler warnings? If you're using *lint*, or a similar diagnostic tool, does the code pass all tests? Does the code pass your unit tests? If you've skipped any of these steps, you're missing an opportunity to easily detect bugs.
- Have you stepped through all new code using a debugger, focusing not only on the code, but also on the data flowing through that code? This is perhaps the best approach to catching bugs in your implementations.
- Have you "cleaned up" any code? If so, have you tested the code? Have you stepped through the code in a debugger? Remember, code that has been cleaned up is actually new code that must be thoroughly tested.
- Should you write a unit test for the new code?

DEBUGGING

You should review the questions below each time you have to track down a reported bug.

- Were you able to find the reported bug? If not, remember that bugs don't just go away; either they're hiding, or they have been fixed already. To determine which is true, you should look for the bug in the same version of the code in which the bug was reported.
- Have you found the true cause of the bug or merely a symptom of the bug? Be sure to track down the cause of the bug.
- How could this bug have been prevented? Come up with a precise guideline that could prevent this bug in the future.
- How could this bug have been detected automatically? Would an assertion catch it? What about some *DEBUG* code? What changes in your coding practices or process would help?

It's not enough that your functions be bug-free; functions must be easy to use without introducing unexpected bugs. If bug rates are to be reduced, each function needs to have one well-defined purpose, to have explicit single-purpose inputs and outputs, to be readable at the point where it is called, and ideally to never return an error condition. Functions with these attributes are easy to validate using assertions and debug code, and they minimize the amount of error handling code that must be written.

Given the numerous implementation possibilities for a given function, it should come as no surprise that some implementations will be more error-prone than others. The key to writing robust functions is to exchange risky algorithms and language idioms for alternatives that have proven to be comparably efficient yet much safer. At one extreme this can mean using unambiguous data types; at the other it can mean tossing out an entire design simply because it would be difficult, or impossible, to test.

Some programming practices are so risky they should never be used. Most such practices are obviously risky, but some seem quite safe, even desirable, because they fill a need without apparent hazard. These treacherous coding practices are the wolves in sheep's clothing. Why shouldn't you reference memory you've just released? Why is it risky to pass data in global or static storage? Why should you avoid parasitic functions? Why it is unwise to rely on every nit-picky detail outlined in the ANSI standard?

A programmer can follow every guideline in this book, but without the proper attitude and a set of good programming habits, writing bug-free code will be much harder than it needs to be. If a programmer believes that a bug can simply "go away," or that fixing bugs "later" won't be harmful to the product, bugs will persist. If a programmer regularly "cleans up" code, allows unnecessary flexibility in functions, welcomes every "free" feature that pops out of a design, or simply "tries" haphazard solutions to problems hoping to hit upon something that works, writing bug-free code will be an uphill battle. Having a good set of habits and attitudes is possibly the most important requirement for consistently writing bug-free code.

Appendix C

Principles

"As I knew, or thought I knew, what was right and wrong,
I did not see why I might not always
do the one and avoid the other.
But I soon found I had undertaken
a task of more difficulty than I had imagined."
—Benjamin Franklin

1. THE PRINCIPLED PROGRAMMER understands a principle well enough to form an opinion about it. (Page 1)
2. Free the future: reuse code. (Page 18)
3. Design and abide by interfaces as though you were the user. (Page 20)
4. Declare data fields `protected`. (Page 20)
5. Test assertions in your code. (Page 26)
6. Maintaining a consistent interface makes a structure useful. (Page 39)
7. Recursive structures must make "progress" toward a "base case." (Page 60)
8. When manipulating references, draw pictures. (Page 105)
9. Every public method of an object should leave the object in a consistent state. (Page 108)
10. Symmetry is good. (Page 111)
11. Test the boundaries of your structures and methods. (Page 114)
12. Question asymmetry. (Page 117)
13. Understand the complexity of the structures you use. (Page 142)
14. Never modify a data structure while an associated **Enumeration** is live. (Page 156)
15. Assume that values returned by iterators are read-only. (Page 162)
16. Declare parameters of overriding methods with the most general types possible. (Page 170)

17. Avoid multiple casts of the same object by assigning the value to a temporary variable. (Page 172)
18. Consider your code from different points of view. (Page 181)
19. Don't let opposing references show through the interface (Page 196)
20. Use wrappers to provide a consistent interface to recursive structures. (Page 201)
21. Write methods to be as general as possible. (Page 211)
22. Avoid unnaturally extending a natural interface. (Page 227)
23. Seek structures with reduced friction. (Page 228)
24. Declare object-independent functions `static`. (Page 230)
25. Provide a method for hashing the objects you implement. (Page 279)
26. Equivalent objects should return equal hash codes. (Page 279)
27. Make it `public` and they will use it. (Page 346)
28. Fight imperfection. (Page 347)

At least 28 is perfect!

In: D.A. Bailey. Data Structures in Java for the Principled Programmer. McGraw-Hill. 1999.

B. Kernighan, R. Pike :
"*The Practice of Programming*".
Addison-Wesley, 1999.

Appendix: Collected Rules

Each truth that I discovered became a rule that served me afterwards in the discovery of others.

René Descartes, *Le Discours de la Méthode*

Several chapters contain rules or guidelines that summarize a discussion. The rules are collected here for easy reference. Bear in mind that each was presented in a context that explains its purpose and applicability.

Style

Use descriptive names for globals, short names for locals.
Be consistent.
Use active names for functions.
Be accurate.
Indent to show structure.
Use the natural form for expressions.
Parenthesize to resolve ambiguity.
Break up complex expressions.
Be clear.
Be careful with side effects.
Use a consistent indentation and brace style.
Use idioms for consistency.
Use else-ifs for multi-way decisions.
Avoid function macros.
Parenthesize the macro body and arguments.
Give names to magic numbers.
Define numbers as constants, not macros.
Use character constants, not integers.
Use the language to calculate the size of an object.
Don't belabor the obvious.

Comment functions and global data.
Don't comment bad code, rewrite it.
Don't contradict the code.
Clarify, don't confuse.

Interfaces

Hide implementation details.
Choose a small orthogonal set of primitives.
Don't reach behind the user's back.
Do the same thing the same way everywhere.
Free a resource in the same layer that allocated it.
Detect errors at a low level, handle them at a high level.
Use exceptions only for exceptional situations.

Debugging

Look for familiar patterns.
Examine the most recent change.
Don't make the same mistake twice.
Debug it now, not later.
Get a stack trace.
Read before typing.
Explain your code to someone else.
Make the bug reproducible.
Divide and conquer.
Study the numerology of failures.
Display output to localize your search.
Write self-checking code.
Write a log file.
Draw a picture.
Use tools.
Keep records.

Testing

Test code at its boundaries.
Test pre- and post-conditions.
Use assertions.
Program defensively.
Check error returns.
Test incrementally.
Test simple parts first.
Know what output to expect.
Verify conservation properties.
Compare independent implementations.
Measure test coverage.
Automate regression testing.
Create self-contained tests.

Performance

Automate timing measurements.
Use a profiler.
Concentrate on the hot spots.
Draw a picture.
Use a better algorithm or data structure.
Enable compiler optimizations.
Tune the code.
Don't optimize what doesn't matter.
Collect common subexpressions.
Replace expensive operations by cheap ones.
Unroll or eliminate loops.
Cache frequently-used values.
Write a special-purpose allocator.
Buffer input and output.
Handle special cases separately.
Precompute results.
Use approximate values.
Rewrite in a lower-level language.
Save space by using the smallest possible data type.
Don't store what you can easily recompute.

Portability

Stick to the standard.
Program in the mainstream.
Beware of language trouble spots.
Try several compilers.
Use standard libraries.
Use only features available everywhere.
Avoid conditional compilation.
Localize system dependencies in separate files.
Hide system dependencies behind interfaces.
Use text for data exchange.
Use a fixed byte order for data exchange.
Change the name if you change the specification.
Maintain compatibility with existing programs and data.
Don't assume ASCII.
Don't assume English.