The Powers of Ten
Rules for Developing Safety Critical Code [*]

Gerard J. Holzmann

2006

Contents

[*]The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Most serious software development projects use coding guidelines. These guidelines are meant to state what the ground rules are for the software to be written: how it should be structured and which language features should and should not be used. Curiously, there is little consensus on what a good coding standard is. Among the many that have been written there are remarkable few patterns to discern, except that each new document tends to be longer than the one before it. The result is that most existing guidelines contain well over a hundred rules, sometimes with questionable justification. Some rules, especially those that try to stipulate the use of white-space in programs, may have been introduced by personal preference; others are meant to prevent very specific and unlikely types of error from earlier coding efforts within the same organization. Not surprisingly, the existing coding guidelines tend to have little effect on what developers actually do when they write code. The most dooming aspect of many of the guidelines is that they rarely allow for comprehensive tool-based compliance checks. Tool-based checks are important, since it is often infeasible to manually review the hundreds of thousands of lines of code that are written for larger applications.

The benefit of existing coding guidelines is therefore often small, even for critical applications. A verifiable set of well-chosen coding rules could, however, make critical software components more thoroughly analyzable, for properties that go beyond compliance with the set of rules itself. To be effective, though, the set of rules has to be small, and must be clear enough that it can easily be understood and remembered. The rules will have to be specific enough that they can be checked mechanically. To put an easy upper-bound on the number of rules for an effective guideline, I will argue that we can get significant benefit by restricting to no more than ten rules. Such a small set, of course, cannot be all-encompassing, but it can give us a foothold to achieve measurable effects on software reliability and verifiability. To support strong checking, the rules are somewhat strict–one might even say Draconian. The trade-off, though, should be clear. When it really counts, especially in the development of safety critical code, it may be worth going the extra mile and living within stricter limits than may be desirable. In return, we should be able to demonstrate more convincingly that critical software will work as intended.

Ten Rules for Safety Critical Coding

The choice of language for a safety critical code is in itself a key consideration, but we will not debate it much here. At many organizations, JPL included, critical code is written in C. With its long history, there is extensive tool support for this language, including strong source code analyzers, logic model extractors, metrics tools, debuggers, test support tools, and a choice of mature, stable compilers. For this reason, C is also the target of the majority of coding guidelines that have been developed. For fairly pragmatic reasons, then, our coding rules primarily target C and attempt to optimize our ability to more thoroughly check the reliability of critical applications written in C.

The following rules may provide benefit, especially if the low number means that developers will actually adhere to them. Each rule is followed with a brief rationale for its inclusion.

Rule 1

Restrict all code to very simple control flow constructs–do not use *goto* statements, *setjmp* or *longjmp* constructs, and direct or indirect *recursion*.

Rationale

Simpler control flow translates into stronger capabilities for verification and often results in improved code clarity. The banishment of recursion is perhaps the biggest surprise here. Without recursion, though, we are guaranteed to have an acyclic function call graph, which can be exploited by code analyzers, and can directly help to prove that all executions that should be bounded are in fact bounded. (Note that this rule does not require that all functions have a single point of return–although this often also simplifies control flow. There are enough cases, though, where an early error return is the simpler solution.)

Rule 2

All loops must have a fixed upper-bound. It must be trivially possible for a checking tool to *prove* statically that a preset upper-bound on the number of iterations of a loop cannot be exceeded. If the loop-bound cannot be proven statically, the rule is considered violated.

Rationale

The absence of recursion and the presence of loop bounds prevents runaway code. This rule does not, of course, apply to iterations that are meant to be non-terminating (e.g., in a process scheduler). In those special cases, the reverse rule is applied: it should be statically provable that the iteration *cannot* terminate. One way to support the rule is to add an explicit upper-bound to all loops that have a variable number of iterations (e.g., code that traverses a linked list). When the upper- bound is exceeded an assertion failure is triggered, and the function containing the failing iteration returns an error. (See Rule 5 about the use of assertions.)

Rule 3

Do not use dynamic memory allocation after initialization.

Rationale

This rule is common for safety critical software and appears in most coding guidelines. The reason is simple: memory allocators, such as *malloc*, and garbage collectors often have unpredictable behavior that can significantly impact performance. A notable class of coding errors also stems from mishandling of memory allocation and free routines: forgetting to free memory or continuing to use memory after it was freed, attempting to allocate more memory than physically available, overstepping boundaries on allocated memory, etc. Forcing all applications to live within a fixed, pre-allocated, area of memory can eliminate many of these problems and make it easier to verify memory use. Note that the only way to dynamically claim memory in the absence of memory allocation from the heap is to use stack memory. In the absence of recursion (Rule 1), an upper-bound on the use of stack memory can derived statically, thus making it possible to prove that an application will always live within its pre-allocated memory means.

Rule 4

No function should be longer than what can be printed on a single sheet of paper in a standard reference format with one line per statement and one line per declaration. Typically, this means no more than about 60 lines of code per function.

Rationale

Each function should be a logical unit in the code that is understandable and verifiable as a unit. It is much harder to understand a logical unit that spans multiple screens on a computer display or multiple pages when printed. Excessively long functions are often a sign of poorly structured code.

Rule 5

The *assertion density* of the code should average to a minimum of two assertions per function. Assertions are used to check for anomalous conditions that should never happen in real-life executions. Assertions must always be side-effect free and should be defined as Boolean tests. When an assertion fails, an explicit recovery action must be taken, e.g., by returning an error condition to the caller of the function that executes the failing assertion. Any assertion for which a static checking tool can prove that it can never fail or never hold violates this rule. (I.e., it is not possible to satisfy the rule by adding unhelpful "`assert(true)`" statements.)

Rationale

Statistics for industrial coding efforts indicate that unit tests often find at least one defect per 10 to 100 lines of code written. The odds of intercepting defects increase with assertion density. Use of assertions is often also recommended as part of a strong defensive coding strategy. Assertions can be used to verify pre-and post-conditions of functions, parameter values, return values of functions, and loop- invariants. Because assertions are side-effect free, they can be selectively disabled after testing in performance-critical code.

A typical use of an assertion would be as follows:

```
if (!c_assert(p >= 0) == true) {
        return ERROR;
}
```

with the assertion defined as follows:

```
#define c_assert(e) ((e) ? (true) : \
    tst_debugging("%s,%d: assertion '%s' failed\n", \
    __FILE__, __LINE__, #e), false)
```

In this definition, `__FILE__` and `__LINE__` are predefined by the macro preprocessor to produce the filename and line-number of the failing assertion. The syntax `#e` turns the assertion condition `e` into a string that is printed as part of the error message. In code destined for an embedded processor there is of course no place to print the error message itself–in that case, the call to `tst_debugging` is turned into a no-op, and the assertion turns into a pure Boolean test that enables error recovery from anomalous behavior.

Rule 6

Data objects must be declared at the smallest possible level of scope.

Rationale

This rule supports a basic principle of data-hiding. Clearly if an object is not in scope, its value cannot be referenced or corrupted. Similarly, if an erroneous value of an object has to be diagnosed, the fewer the number of statements where the value could have been assigned; the easier it is to diagnose the problem. The rule discourages the re-use of variables for multiple, incompatible purposes, which can complicate fault diagnosis.

Rule 7

The return value of non-void functions must be checked by each calling function, and the validity of parameters must be checked inside each function.

Rationale

This is possibly the most frequently violated rule, and therefore somewhat more suspect as a general rule. In its strictest form, this rule means that even the return value of `printf` statements and file `close` statements must be checked. One can make a case, though, that if the response to an error would rightfully be no different than the response to success, there is little point in explicitly checking a return value. This is often the case with calls to `printf` and `close`. In cases like these, it can be acceptable to explicitly cast the function return value to `(void)`–thereby indicating that the programmer explicitly and not accidentally decides to ignore a return value. In more dubious cases, a comment should be present to explain why a return value is irrelevant. In most cases, though, the return value of a function should not be ignored, especially if error return values must be propagated up the function call chain. Standard libraries famously violate this rule with potentially grave consequences. See, for instance, what happens if you accidentally execute `strlen(0)`, or `strcat(s1, s2, -1)` with the standard C string library–it is not pretty. By keeping the general rule, we make sure that exceptions must be justified, with mechanical checkers flagging violations. Often, it will be easier to comply with the rule than to explain why non-compliance might be acceptable.

Rule 8

The use of the preprocessor must be limited to the inclusion of header files and simple macro definitions. Token pasting, variable argument lists (ellipses), and recursive macro calls are not allowed. All macros must expand into complete syntactic units. The use of conditional compilation directives is often also dubious, but cannot always be avoided. This means that there should rarely be justification for more than one or two conditional compilation directives even in large software development efforts, beyond the standard boilerplate that avoids multiple inclusion of the same header file. Each such use should be flagged by a tool-based checker and justified in the code.

Rationale

The C preprocessor is a powerful obfuscation tool that can destroy code clarity and befuddle many text based checkers. The effect of constructs in unrestricted preprocessor code can be extremely hard to decipher, even with a formal language definition in hand. In a new implementation of the C preprocessor, developers often have to resort to using earlier implementations as the referee for interpreting complex defining language in the C standard. The rationale for the caution against conditional compilation is equally important. Note that with just ten conditional compilation directives, there could be up to 2^{10} possible versions of the code, each of which would have to be tested–causing a huge increase in the required test effort.

Rule 9

The use of pointers should be restricted. Specifically, no more than one level of dereferencing is allowed. Pointer dereference operations may not be hidden in macro definitions or inside `typedef` declarations. Function pointers are not permitted.

Rationale

Pointers are easily misused, even by experienced programmers. They can make it hard to follow or analyze the flow of data in a program, especially by tool-based static analyzers. Function pointers, similarly, can seriously restrict the types of checks that can be performed by static analyzers and should only be used if there is a strong justification for their use, and ideally alternate means are provided to assist tool-based checkers determine flow of control and function call hierarchies. For instance, if function pointers are used, it can become impossible for a tool to prove absence of recursion, so alternate guarantees would have to be provided to make up for this loss in analytical capabilities.

Rule 10

All code must be compiled, from the first day of development, with *all* compiler warnings enabled at the compiler's most pedantic setting. All code must compile with these setting without any warnings. All code must be checked daily with at least one, but preferably more than one, state-of-the-art static source code analyzer and should pass the analyses with zero warnings.

Rationale

There are several very effective static source code analyzers on the market today, and quite a few freeware tools as well.[1]There simply is no excuse for any software development effort not to make use of this readily available technology. It should be considered routine practice, even for non-critical code development. The rule of zero warnings applies even in cases where the compiler or the static analyzer gives an erroneous warning: if the compiler or the static analyzer gets confused, the code causing the confusion should be rewritten so that it becomes more trivially valid. Many developers have been caught in the assumption that a warning was surely invalid, only to realize much later that the message was in fact valid for less obvious reasons. Static analyzers have somewhat of a bad reputation due to early predecessors, such as *lint*, that produced mostly invalid messages, but this is no longer the case. The best static analyzers today are fast, and they produce selective and accurate messages. Their use should not be negotiable at any serious software project.

The first two rules guarantee the creation of a clear and transparent control flow structure that is easier to build, test, and analyze. The absence of dynamic memory allocation, stipulated by the third rule, eliminates a class of problems related to the allocation and freeing of memory, the use of stray pointers, etc. The next few rules (4 to 7) are fairly broadly accepted as standards for good coding style. Some benefits of other coding styles that have been advanced for safety critical systems, e.g., the discipline of "design by contract" can partly be found in rules 5 to 7.

These ten rules are being used experimentally at JPL in the writing of mission critical

[1]For an overview see, for instance, `http://spinroot.com/static/index.html`

software, with encouraging results. After overcoming a healthy initial reluctance to live within such strict confines, developers often find that compliance with the rules does tend to benefit code clarity, analyzability, and code safety. The rules lessen the burden on the developer and tester to establish key properties of the code (e.g., termination or boundedness, safe use of memory and stack, etc.) by other means. If the rules seem Draconian at first, bear in mind that they are meant to make it possible to check code where very literally your life may depend on its correctness: code that is used to control the airplane that you fly on, the nuclear power plant a few miles from where you live, or the spacecraft that carries astronauts into orbit. The rules act like the seat-belt in your car: initially they are perhaps a little uncomfortable, but after a while their use becomes second-nature and not using them becomes unimaginable.