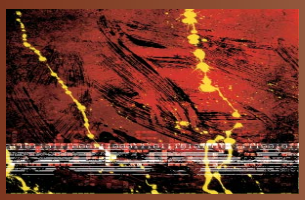
The Power of 10: Rules for Developing Safety-Critical Code

Gerard J. Holzmann
NASA/JPL Laboratory for Reliable Software



Adhering to a set of 10 verifiable coding rules can make the analysis of critical software components more reliable.

Most serious software development projects use coding guidelines. These guidelines are meant to define the ground rules 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 coding guidelines that have been written, there are remarkably 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 100 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 errors 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 because manually reviewing the hundreds of thousands of lines of code that are written for larger applications is often infeasible.

Existing coding guidelines therefore offer limited benefit, even for critical applications. A verifiable set of well-chosen coding rules could, however, assist in analyzing critical software components for properties that go well beyond compliance with the set of rules itself. To be effective, though, the set of rules must be small, and it must be clear enough that users can easily understand and remember it. In addition, the rules must be specific enough that users can check them thoroughly and mechanically.

To put an upper bound on the number of rules, I will argue that restricting the set to no more than 10 rules will provide an effective guideline. Although such a small set of rules cannot be all-encompassing, following it can achieve measurable effects on software reliability and verifiability.

To support strong checking, the rules I will propose are somewhat strict—some might even say draconian. The tradeoff, though, should be clear. When it really counts, especially in the development of safety-critical code, working within stricter limits can be worth the extra effort. In return, it should be possible to demonstrate more convincingly that critical software will work as intended.

SAFETY-CRITICAL CODING RULES

The choice of language for safety-critical code is in itself a key consideration. At many organizations, JPL included, developers write most code 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 existing coding guidelines. For fairly pragmatic reasons, then, the following 10 rules primarily target C and attempt to optimize the ability to more thoroughly check the reliability of critical applications written in C.

These rules might prove to be beneficial, especially if the small number means that developers will actually adhere to them.

Rule 1: Restrict all code to very simple control flow constructs—do not use *goto* statements, *setjmp* or *longjmp* constructs, or direct or indirect recursion.
Rationale: Simpler control flow translates into stronger capabilities for analysis and often results in improved code clarity. Banishing recursion is perhaps the biggest surprise here. Avoiding recursion results in having an acyclic function call graph, which code

analyzers can exploit to prove limits on stack use and boundedness of executions. Note that this rule does not require that all functions have a single point of return, although this often also simplifies control flow. In some cases, though, an early error return is the simpler solution.

Rule 2: Give all loops a fixed upper bound. It must be trivially possible for a checking tool to prove statically that the loop cannot exceed a preset upper bound on the number of iterations. If a tool cannot prove the loop bound 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 nonterminating—for example, in a process scheduler. In those special cases, the reverse rule is applied: It should be possible for a checking tool to prove statically that the iteration *cannot* terminate.

One way to comply with this rule is to add an explicit upper bound to all loops that have a variable number of iterations—for example, code that traverses a linked list. When the loop exceeds the upper bound, it must trigger an assertion failure, and the function containing the failing iteration should return an error.

Rule 3: Do not use dynamic memory allocation after initialization.
Rationale: This rule appears in most coding guidelines for safety-critical software. 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 the 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, and so on. Forcing all applications to live within a fixed, preallocated 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, an upper bound on the use of stack memory can be derived statically, thus making it possible to prove that an application will always live within its resource bounds.

Following a small set of rules can achieve measurable effects on software reliability and verifiability.

Rule 4: No function should be longer than what can be printed on a single sheet of paper in a standard 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 pages. Excessively long functions are often a sign of poorly structured code.

Rule 5: The code's assertion density should average to minimally two assertions per function. Assertions must be used to check for anomalous conditions that should never happen in real-life executions. Assertions must be side-effect free and should be defined as Boolean tests. When an assertion fails, an explicit recovery action must be taken such as 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.
Rationale: Statistics for industrial coding efforts indicate that unit tests often find at least one defect per 10 to 100 lines of written code. The odds of intercepting defects increase significantly with increasing assertion density. Using assertions is often recommended as part of a strong defensive coding strategy. Developers can use assertions to verify pre- and postconditions of functions, parameter values, return values of functions, and loop invariants. Because the proposed assertions are side-effect free, they can be selectively disabled after testing in performance-critical code.

Rule 6: Declare all data objects 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, other modules cannot reference or corrupt its value. Similarly, if a tester must diagnose an object's erroneous value, the fewer the number of statements where the value could have been assigned, the easier it is to diagnose the problem. The rule also discourages the reuse of variables for multiple, incompatible purposes, which can complicate fault diagnosis.

Rule 7: Each calling function must check the return value of nonvoid functions, and each called function must check the validity of all parameters provided by the caller.
Rationale: This is possibly the most frequently violated rule, and therefore it is somewhat more suspect for inclusion 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. Yet, if the response to an error would 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, explicitly casting the function return value to *(void)* can be acceptable, thereby indicating that the programmer explicitly and not accidentally decided to ignore a return value.

In more dubious cases, a comment should be offered to explain why a return value can be considered irrelevant. In most cases, though, a function's return value should not be

ignored, especially if the function should propagate an error return value up the function call chain.

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 must be kept to a minimum.
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 a new implementation of the C preprocessor, developers often must resort to using earlier implementations to interpret complex defining language in the C standard. The rationale for the caution against conditional compilation is equally important. With just 10 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. The use of conditional compilation cannot always be avoided, but even in large software development efforts there is rarely justification for more than one or two such directives, beyond the standard boilerplate that avoids multiple inclusions of the same header file. A tool-based checker should flag each use and each use should be justified in the code.

Rule 9: The use of pointers must be restricted. Specifically, no more than one level of dereferencing should be used. 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 analyzers. Similarly, function pointers should be used only if there is a very strong justification for doing so because they can seriously restrict the types of automated checks that code checkers can perform. For example, if function pointers are used, it can become impossible for a tool to prove the absence of recursion, requiring alternate guarantees to make up for this loss in checking power.

Rule 10: All code must be compiled, from the first day of development, with all compiler warnings enabled at the most pedantic setting available. All code must compile without warnings. All code must also be checked daily with at least one, but preferably more than one, strong static source code analyzer and should pass all analyses with zero warnings.
Rationale: There are several extremely effective static source code analyzers on the market today, and quite a few freeware tools as well. There simply is no excuse for any software development effort not to use this readily available technology. It should be considered routine practice, even for non-critical code development.

The rule of zero warnings applies even when the compiler or the static analyzer gives an *erroneous* warning: If the compiler or analyzer gets confused, the code causing the confusion should be rewritten. 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 versions that produced mostly invalid messages, but this is no longer the case. The best static analyzers today are fast, and they produce accurate messages. Their use should not be negotiable on any serious software project.

FOLLOWING THE RULES

The first few rules from this set 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 are fairly broadly accepted as standards for good coding style. Other rules secure some of the benefits of stronger coding styles that have been advanced for safety-critical systems such as the "design by contract" discipline.

Developers are currently using this rule set experimentally at JPL to write mission-critical 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 safety. The rules lessen the burden on developers and testers to use other means to establish key properties of code such as termination or boundedness and safe use of memory and stack.

If these rules seem draconian at first, bear in mind that they are meant to make it possible to check safety-critical code where human lives can very literally depend on its correctness. The rules are like the seat belts in a car: Initially, using them is perhaps a little uncomfortable, but after a while it becomes second nature, and not using them is unimaginable.

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.

***Gerard J. Holzmann** is a Principal Computer Scientist at NASA's Jet Propulsion Laboratories, where he leads the Laboratory for Reliable Software. Contact him at gholzmann@acm.org.*

Editor: Michael G. Hinchey, NASA Software Engineering Laboratory at NASA Goddard Space Flight Center and Loyola College in Maryland; michael.g.hinchey@nasa.gov