WHEN I FIRST learned to code, in what now seems more like the dark ages, the first step in program development was to take a coding sheet and write out the program’s design. Then, someone had to sign off on that design, to make sure it wasn’t too obviously wrong. Next, I had to wait my turn at a punch-card machine to type the program line by line onto punch cards (see Figure 1). When that was complete, I placed the card deck in a little bin with my name on it. In due time, a computer operator would pick up the deck and feed it into a large mainframe computer. The computer center proudly offered a rapid 24-hour turnaround on jobs. So, the very next day I could collect the printed output and discover all the things that had gone wrong. This process then had to be repeated, day after day, until the program was finally right.

As painful as this process might seem today, it taught me to be very careful with my code. Just getting the program to execute was a time-consuming battle with syntax errors or with mistakes in the also-required “job control” cards that told the behemoth machine how to execute a program. And then the real frustration would begin. No matter how certain I was that my program was correct, the machine would manage to find something to complain about. The delays were never the machine’s fault. It was just very good at confronting me with my own fallibility.

How are things different now? Thank goodness, the long wait to find out how we’re messing things up is gone, now that we have more compute power in our watch than a roomful of hardware in the dark ages. But we’re still human, and we still make mistakes. We can just make and discover our mistakes much faster than before.

Most compilers have built-in checks that warn against common types of coding flaws. Static source code analysis tools also excel at finding bugs and revealing risky coding patterns. Modern coding standards similarly include a lot of rules that...
can prevent common mistakes. But now that our tools have gotten so efficient in pointing out where we’re taking unnecessary risks, we come up against some other very human characteristics: programmers, and perhaps humans in general, don’t like following rules. In fact, we take pride in finding creative ways around them.

Dodging rules, or finding clever ways around them, happens in every field. For instance, in 2013, the National Hockey League forbade players from taking off their helmets when a fight breaks out during a game. So, players simply took off each other’s helmets before enthusiastically engaging in a fight. Similarly, after a town issued an ordinance forbidding vendors to sell drinks during street festivals, a quick-thinking vendor figured out that he could still sell a peanut for a dollar and offer a free bottle of water with each purchase. Even the US Congress isn’t immune to this type of creative subversion. In 2011, it declared that pizza could count as a vegetable in school lunches, thus letting schools get around the stricter rules for lunches that were about to go into effect. We all love to outsmart the system, and that’s just as true when it comes to writing code.

**Outsmarting the System**

Near the top of the list of the most common types of mistakes programmers make is the unintentional use of uninitialized data. In C, if you declare a global variable and forget to initialize it, its initial value is guaranteed to be zero. That might be the right value, but the compiler can’t really tell, so it won’t issue a warning. If, however, you omit the initialization in the declaration of a function local variable, your variable’s value will be undefined. It will be whatever happens to be in memory at the stack location where the automatic variable is stored. If the first access to that variable tries to retrieve its value, the result will likely be wrong (see Figure 2). A good compiler will therefore issue a warning for such cases.

The root cause for the accidental use of an uninitialized variable is often the existence of an execution path the developer didn’t realize existed. To fix it generally requires rewriting the algorithm to ensure the unsuspected execution path is safe. That requires thought.

But there’s also another, quicker way to remove the warning. You can explicitly initialize the local variable to any value at all in its declaration, and the warning will go away. The code, though, isn’t necessarily any better than it was before, and it might actually be worse. You’ve only succeeded in hiding it better from the tools that are trying to protect you from yourself. In Figure 2, the result of initializing the local variable x to the value 1 would be a run-time error when you later try to divide the sum of x and y by their difference.

**The Gentle Art of Sidestepping**

There is, of course, no shortage of examples of programmers sidestepping rules and of the risks this can create. For instance, safety-critical applications often have a rule limiting the use of conditional compilation directives. The one exception that’s generally allowed is to use a conditional directive to prevent the repeated processing of header files. The reason for frowning on conditional compilation directives is that their unrestricted use can dramatically increase the number of ways the code can be compiled and thus the number of ways it should be tested. With just 10 such directives, the code can be compiled in up to $2^{10}$ (1,024) ways, but it would most likely be tested thoroughly for only one. This leaves all other possible variants at risk, including the one that might be called into action the very last moment before a public release.

Finding violations of this coding rule isn’t hard: a scan of the source files with a simple script that looks for the telltale `#ifdef` and `#ifndef` keywords suffices. But it’s easy to defeat this type of check if the resulting warnings become too annoying. For instance, a developer can first define a macro that’s statically defined to be either true or false. Then, he or she can employ a Boolean test of that value in a standard `if-then-else` construct in the code. This achieves the same effect of the conditional compilation directive, but with the benefit that the checker can’t detect it. But the vulnerability is still there.

Another trick is remarkably popular to defeat compiler warnings about unused parameters to
functions. You could define a parameter that’s meant to be used when the code for a function is complete, but that isn’t used when earlier versions of the code are tested. It isn’t too hard to fool the compiler into thinking that the parameter is used, to avoid those pesky warnings that the compiler otherwise generates.

Figure 3 shows a common method in which the programmer simply adds a line of code that evaluates the parameter’s value and then discards it explicitly with a cast to (void). Most compilers accept this quietly and don’t generate code for the added statement, and you score a quiet victory over the machine. The danger is that you might forget to add the originally intended code that properly uses the parameter, because all now seems to be well with the function as written.

### Manipulating Metrics

According to Goodhart’s law (named after the economist Charles Goodhart), “When a measure becomes a target, it ceases to be a good measure.” Code metrics are particularly easy to manipulate in your favor. Consider, for instance, what would happen if you reward software developers solely on the basis of the number of lines of code they write per day. “Productivity” will likely increase dramatically in a short amount of time. Similarly, if you evaluate a test suite only on the basis of a blind measurement of code coverage, you’re likely to inspire the creation of test suites that excel at executing lines of code, without actually testing them.

Another metric that program managers like is comment density. Large projects often require every function to have a comment header based on a predefined template that includes entries for defining the parameters to the function and the return value it’s meant to compute. More often than not, though, the template is just copied into the code unchanged. This satisfies the formal rule that the template is present but doesn’t actually provide any useful information about the function’s purpose or use. And who hasn’t seen those eminently superfluous comments of the type “assign a new value to $x$” or “subtract one from the value of $y$.” It’s just like the kids in high school when they figure out that using words such as “magnificent” or “inordinate” increases the writing score they get from automatic grading tools in their English class. Before you know it, just about everything becomes “inordinately magnificent.”

### Ducking Compliance

Figure 4 shows a remarkable piece of code that illustrates what can happen if you move perilously beyond a language definition’s boundaries. (Xi Wang and his colleagues described a similar example in 2013.) The C language standard explicitly enumerates all implementation-defined and formally undefined language features. Nonetheless, exploiting some of those features is for some the ulti-
mate test of cool. Be warned—if you do so, you’ll be at the mercy of the authors of the specific version of the compiler you’re using, running on a specific hardware platform.

If you compile and execute the code from Figure 4 without optimization, the result will be \( x < 0 \) and \( -x < 0 \), which says that the integer variable \( x \) is negative and the negation of \( x \) is also negative. That would surprise most math majors, but it’s the result of having initialized \( x \) to the minimum value that can be represented in an integer.

So, what’s going on? For 32-bit signed integers, the value of \( \text{INT\_MIN} \) is \(-2^{31} \) (\(-2,147,483,648\)). The two’s-complement binary representation encodes this as a single one followed by 31 zeros. On the other end of the spectrum, the maximum value representable in a signed integer is \( 2^{31} - 1 \) (2,147,483,647), which is encoded as a single zero followed by 31 ones. If you try to take the negation of \( \text{INT\_MIN} \), like the code in Figure 4 does, you might expect to get 2,147,483,648, but that’s one more than \( \text{INT\_MAX} \). And, through the wonders of the two’s complement, one more than \( \text{INT\_MAX} \) gets the same encoding as \( \text{INT\_MIN} \). So, in two’s complement, the negation of \( \text{INT\_MIN} \) is equal to \( \text{INT\_MIN} \) itself, which explains how the program executes. But does it?

If you compile the code again, but this time enable compiler optimization (for instance, using gcc compiler version 4.6.3), the printed result can change to “everything is OK.”

What gives? Clearly, the program didn’t change, and most likely neither did the representation of integer numbers that the compiler used for storing integer numbers. This is where the smarts of a compiler writer can outsmart those of the programmer. The optimizer can reason that because a true integer number can’t be both negative and positive, it can replace the conditional \( (x < 0 \ \&\& \ -x < 0) \) with \texttt{false}. This optimization forces the program’s execution into the \texttt{else} branch and gives a new result. Whether or not you use optimization, the gcc compiler issues no warnings, so you aren’t alerted to the code’s hidden ambiguity.

The C language standard warns clearly about the use of integer values that can’t be represented in the available bit-width of the machine, noting that the result of such operations is implementation-defined. This means that not every compiler will necessarily produce different results depending on how the code is compiled. When something is implementation-defined, it’s simply up to the compiler writer to decide what to do, and, indeed, compiler writers too.

W

iting reliable code is hard. What’s so interesting about software development is that even the smallest mistake can have the largest consequences. As a comedian once said, “When you dial a wrong number, and you’re off by only one digit, I don’t think you should get a whole different person!” But this is precisely what can happen in software systems. A single misplaced \texttt{break} statement can paralyze the entire long-distance calling network, as it did in the US on 15 January 1990. A single uninitialized variable can make a spacecraft crash, as it did on 3 December 1999 when the Mars Polar Lander failed its landing.

What’s so fascinating is that these types of failure seem completely out of proportion with the flaws that caused them. This is also what makes designing and building reliable software systems so different from designing and building reliable hardware. The best remedy? Keep your code clean and simple and heed the warnings from the tools that are designed to protect you. Either do that or risk having a dumb machine show you it can continue to outsmart you.

References


GERARD J. HOLZMANN works at the Jet Propulsion Laboratory (JPL) on developing stronger methods for software analysis, code review, and testing. He’s an ACM and a JPL fellow and a member of the National Academy of Engineering. Contact him at gholzmann@acm.org.

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.