



mistakes every programmer makes

No matter how long you've been programming, you've made mistakes like these...

When you start programming, you get disillusioned quickly. No longer is the computer the all-infallible perfect machine – “do as I mean, not as I say” becomes a frequent cry. At night, when the blasted hobgoblins finally go to bed, you lie there and ruminate on the errors you made that day, and they're worse than any horror movie. So when the editor of PC Plus asked me to write this article, I reacted with both fear and knowing obedience.

I was confident that I could dash this off in a couple of hours and nip down to the pub without the usual resultant night terrors. The problem with such a request is, well, which language are we talking about? I can't just trot out the top 10 mistakes you could make in C#, Delphi, JavaScript or whatever – somehow my top ten list has to encompass every language. Suddenly, the task seemed more difficult. The hobgoblins started cackling in my head. Nevertheless, here goes...

Writing for the compiler, not for people

1 When they use a compiler to create their applications, people tend to forget that the verbose grammar and syntax required to make programming easier is tossed aside in the process of converting prose to machine code. A compiler doesn't care if you use a single-letter

identifier or a more human-readable one. The compiler doesn't care if you write optimised expressions or whether you envelop sub-expressions with parentheses. It takes your human-readable code, parses it into abstract syntax trees and converts those trees into machine code, or some kind of intermediate language. Your names are by then history.

So why not use more readable or semantically significant identifiers than just *i*, *j* or *x*? These days, the extra time you would spend waiting for the compiler to complete translating longer identifiers is minuscule. However, the much-reduced time it takes you or another programmer to read your source code when the code is expressly written to be self-explanatory, to be more easily understandable, is quite remarkable.

Another similar point: you may have memorised the operator precedence to such a level that you can omit needless parentheses in your expressions, but consider the next programmer to look at your code. Does he? Will he know the precedence of operators in some other language better than this one and thereby misread your code and make invalid assumptions about how it works?

Personally, I assume that everyone knows that multiplication (or division) is done before addition and subtraction, but that's about it. Anything else in an expression and I throw

in parentheses to make sure that I'm writing what I intend to write, and that other people will read what I intended to say. The compiler just doesn't care.

Studies have shown that the proportion of some code's lifecycle spent being maintained is easily five times more than was spent initially writing it. It makes sense to write your code for someone else to read and understand.

Writing big routines

2 Back when I was starting out, there was a rule of thumb where I worked that routines should never be longer than one printed page of fanfold paper – and that included the comment box at the top that was fashionable back then. Since then, and especially in the past few years, methods tend to be much smaller – merely a few lines of code. In essence, just enough code that you can grasp its significance and understand it in a short time. Long methods are frowned upon and tend to be broken up.

The reason is extremely simple: long methods are hard to understand and therefore hard to maintain. They're also hard to test properly. If you consider that testing is a function of the number of possible paths through a method, the longer the method, the more tests you'll have to write and the more involved those tests will have to be. There's

Spotlight on... The art of programming

For mistake 10 in this article, I recommend that a developer keeps up with modern techniques for developing applications. By following these techniques, you can avoid many of the other mistakes I've mentioned.

Obviously, as we discovered at the very beginning, different languages have their own little quirks to avoid, but overall, the best way to avoid making mistakes is to never make them in the first place. You'll only do that through better education. Here's a short reading list of books that I've found invaluable

in helping me become a better overall programmer, no matter which language I'm using today.

McConnell, S – *Code Complete 2*

Hunt A, & Thomas D – *The Pragmatic Programmer*

Gamma, E, Helm R, Johnson R, & J. Vlissides – *Design Patterns*

Fowler, M – *Refactoring*

Beck, K – *Test-Driven Development*

Evans, E – *Domain-Driven Development*

Martin, R – *Agile Software Development* ■

▶ actually a pretty good measurement you can make of your code that indicates how complex it is, and therefore how probable it is to have bugs – the cyclomatic complexity.

Developed by Thomas J. McCabe Sr in 1976, cyclomatic complexity has a big equation linked to it if you're going to run through it properly, but there's an easy, basic method you can use on the fly. Just count the number of 'if' statements and loops in your code. Add 1 and this is the CC value of the method. It's a rough count of the number of execution paths through the code. If your method has a value greater than 10, I'd recommend you rewrite it.

Premature optimisation

3 This one's simple. When we write code, sometimes we have a niggling devil on our shoulder pointing out that this clever code would be a bit faster than the code you just wrote. Ignore the fact that the clever code is harder to read or harder to comprehend; you're shaving off milliseconds from this loop. This is known as premature optimisation. The famous computer scientist Donald Knuth said, "We should forget about small efficiencies, say about 97 per cent of the time: premature optimisation is the root of all evil."

In other words: write your code clearly and cleanly, then profile to find out where the real bottlenecks are and optimise them. Don't try to guess beforehand.

Using global variables

4 Back when I started, lots of languages had no concept of local variables at all and so I was forced to use global variables. Subroutines were available and encouraged but you couldn't declare a variable just for use within that routine – you had to use one that was visible from all your code.

Still, they're so enticing, you almost feel as if you're being green and environmentally conscious by using them. You only declare them once, and use them all over the place, so it seems you're saving all that precious memory. But it's that "using all over the place" that trips you up. The great thing about global variables is that they're visible everywhere. This is also the worst thing about global variables: you

have no way of controlling who changes it or when the variable is accessed. Assume a global has a particular value before a call to a routine and it may be different after you get control back and you don't notice.

Of course, once people had worked out that globals were bad, something came along with a different name that was really a global variable in a different guise. This was the singleton, an object that's supposed to represent something of which there can only be one in a given program. A classic example, perhaps, is an object that contains information about your program's window, its position on the screen, its size, its caption and the like.

The main problem with the singleton object is testability. Because they are global objects, they're created when first used, and destroyed only when the program itself terminates. This persistence makes them extremely difficult to test. Later tests will be written implicitly assuming that previous tests have been run, which set up the internal state of the singleton.

Another problem is that a singleton is a complex global object, a reference to which is passed around your program's code. Your code is now dependent on some other class. Worse than that, it's coupled to that singleton. In testing, you would have to use that singleton. Your tests would then become dependent on its state, much as the problem you had in testing the singleton in the first place. So, don't use globals and avoid singletons.

Not making estimates

5 You're just about to write an application. You're so excited about it that you just go ahead and start designing and writing it. You release and suddenly you're beset with performance issues, or out-of-memory problems. Further investigations show that, although your design works well with small number of users, or records, or items, it does not scale – think of the early days of Twitter for a good example. Or it works great on your super-duper developer 3GHz PC with 8GB of RAM and an SSD, but on a run-of-the-mill PC, it's slower than a Greenland glacier in January.

Part of your design process should have been some estimates, some back-

of-the-envelope calculations. How many simultaneous users are you going to cater for? How many records? What response time are you targeting? Try to provide estimates to these types of questions and you'll be able to make further decisions about techniques you can build into your application, such as different algorithms or caching. Don't run pell-mell into development – take some time to estimate your goals.

Off by one

6 This mistake is made by everyone, regularly, all the time. It's writing a loop with an index in such a way that the index incremented once too often or once too little. Consequently, the loop is traversed an incorrect number of times. If the code in the loop is visiting elements of an array one by one, a non-existent element of the array may be accessed – or, worse, written to – or an element may be missed altogether.

One reason why you might get an off-by-one error is forgetting whether indexes for array elements are zero-based or one-based. Some languages even have cases where some object is zero-based and others where the assumption is one-based.

There are so many variants of this kind of error that modern languages or their runtimes have features such as 'foreach loops' to avoid the need to count through elements of an array or list. Others use functional programming techniques called map, reduce and filter to avoid the need to iterate over collections.

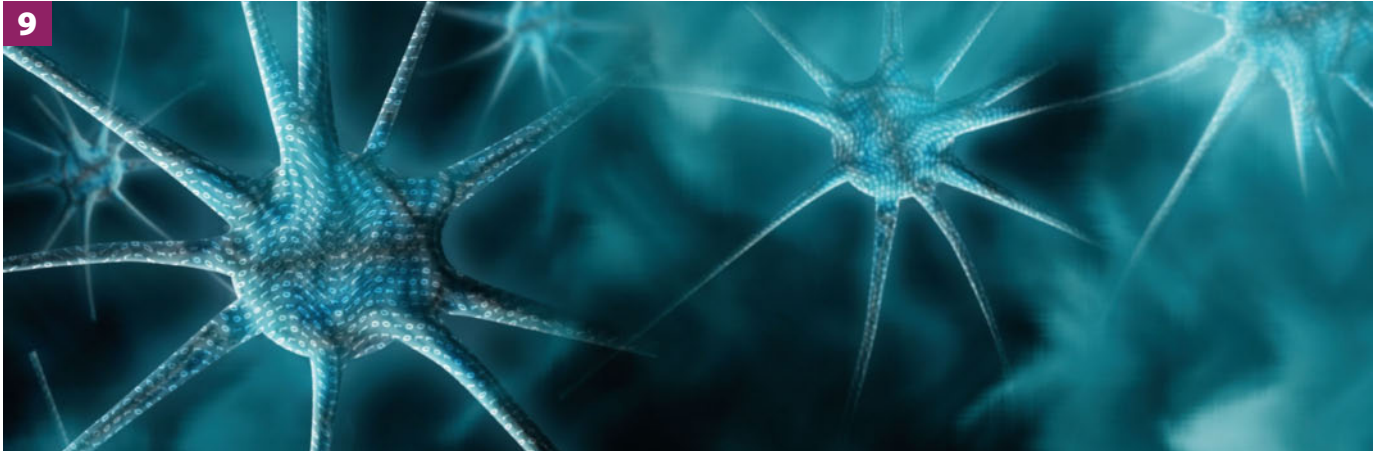
Use modern 'functional' loops rather than iterative loops.

Suppressing exceptions

7 Modern languages use an exception system as an error-reporting technique, rather than the old traditional passing and checking of error numbers. The language incorporates new keywords to dispatch and trap exceptions, using names such as throw, try, finally and catch. The remarkable thing about exceptions is their ability to unwind the stack, automatically returning from nested routines until the exception is trapped and dealt with. No longer



9



do you have to check for error conditions, making your code into a morass of error tests. All in all, exceptions make for more robust software, providing that they're used properly.

Catch is the interesting one: it allows you to trap an exception that was thrown and perform some kind of action based upon the type of the exception. The biggest mistakes programmers make with exceptions are twofold. The first is that the programmer is not specific enough in the type of exception they catch. Catching too general an exception type means that they may be inadvertently dealing with particular exceptions that would be best left to other code, higher up the call chain. Those exceptions would, in effect, be suppressed and possibly lost.

The second mistake is more pernicious: the programmer doesn't want any exceptions leaving their code and so catches them all and ignores them. This is known as the empty catch block. They may think, for example, that only certain types of exceptions might be thrown in his code; ones that they could justifiably ignore. In reality, other deadly runtime exceptions could happen – things such as out-of-memory exceptions, invalid code exceptions and the like, for which the program shouldn't continue running at all.

Tune your exception catch blocks to be as specific as possible.

Storing secrets in plain text

8 A long time ago, I worked in a bank. We purchased a new computer system for the back office to manage some kind of workflow dealing with bond settlements. Part of my job was to check this system to see whether it worked as described and whether it was foolproof. After all, it dealt with millions of pounds daily and then, as now, a company is more likely to be defrauded by an employee than an outsider. After 15 minutes with a rudimentary hex editor, I'd found the administrator's password stored in plain text.

Data security is one of those topics that deserves more coverage than I can justifiably provide here, but you should never, ever store passwords in plain text. The standard for passwords is to store the salted hash of the original password, and then do the same

salting and hashing of an entered password to see if they match.

Here's a handy hint: if a website promises to email you your original password should you forget it, walk away from the site. This is a huge security issue. One day that site will be hacked. You'll read about how many logins were compromised, and you'll swallow hard and feel the panic rising. Don't be one of the people whose information has been compromised and, equally, don't store passwords or other 'secrets' in plain text in your apps.

Not validating user input

9 In the good old days, our programs were run by individuals, one at a time. We grew complacent about user input: after all, if the program crashed, only one person would be inconvenienced – the one user of the program at that time. Our input validation was limited to number validation, or date checking, or other kinds of verification of input. Text input tended not to be validated particularly.

Then came the web. Suddenly your program is being used all over the world and you've lost that connection with the user. Malicious users could be entering data into your program with the express intent of trying to take over your application or your servers.

A whole crop of devious new attacks were devised that took advantage of the lack of checking of user input. The most famous one is SQL injection, although unsanitised user input could precipitate an XSS attack (cross-site scripting) through markup injection.

Test-driven development

Another technique devised to produce well-tested software is test-driven development. In this methodology, the programmer writes what they want the code to do as a unit test before writing the code itself. Once they have the test, they write the smallest amount of code needed to make that test pass. After each portion of tested code is written, the programmer can then refactor the code to eliminate duplication, clean it up and so on, knowing that the changes can be tested with the current unit test suite. In doing this, the programmer is concentrating on what the code should do, and not on how the code does it. ■

Both types rely on the user providing, as part of normal form input, some text that contains either SQL or HTML fragments. If the application does not validate the user input, it may just use it as is and either cause some hacked SQL to execute, or some hacked HTML/JavaScript to be produced. This in turn could crash the app or allow it to be taken over by the hacker.

So, always assume the user is a hacker trying to crash or take over your application and validate or sanitise user input.

Not being up to date

10 All of the previous mistakes have been covered in depth online and in various books. I haven't discovered anything new – they and others have been known for years. These days you have to work pretty hard to avoid coming into contact with various modern design and programming techniques.

I'd say that not spending enough time becoming knowledgeable about programming – and maintaining that expertise – is in fact the biggest mistake that programmers make. They should be learning about techniques such as TDD or BDD, about what SLAP or SOLID means, about various agile techniques. These skills are of equal or greater importance than understanding how a loop is written in your language of choice.

So don't be like them: read McConnell and Beck and Martin and Jeffries and the Gang of Four and Fowler and Hunt & Thomas and so on. Make sure you stay up to date with the art and practice of programming.

And that concludes my top 10 list of mistakes programmers make, no matter what their language stripe. There are others, to be sure, perhaps more disastrous than mine, but I would say that their degree of dread is proportional to the consequences of making them. All of the above were pretty dire for me the last time I made them. If you have further suggestions or calamities of your own, don't hesitate to contact me and let me know. **PCP**

*Julian M Bucknall has worked for companies ranging from TurboPower to Microsoft and is now CTO for Developer Express.
feedback@pcplus.co.uk*