the magazine of the accu

www.accu.org

{cvu}

Volume 29 • Issue 1 • March 2017 • £3

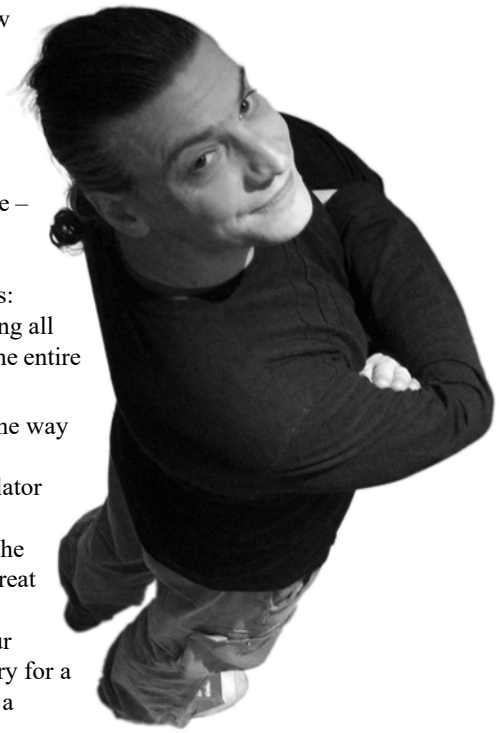
Features

Regulars

{cvu}
Volume 29 Issue 1
March 2017
ISSN 1354-3164
www.accu.org

accu

Unnecessary complexity



In the last issue, I used this space to talk about the Software Crisis of the 1970s and 80s, and how luminaries of the day noted that the solution was the pursuit of simplicity. I also wrote briefly about how that Crisis lives on in the shape of Internet of Things' security (amongst other problems), and I think the same solution applies. I'm not suggesting that security is simple – far from it! Its complexity is not, however, best managed by more complexity. Simplicity, when applied to computer programs, means two things: removing unnecessary complexity, and containing all *necessary* complexity so that it doesn't 'infect' the entire system.

Getting rid of un-needed or useless features is one way of removing complexity. Applying unnecessary technology to stuff isn't even a new idea. Calculator watches, anyone? Perhaps not quite useless, but almost un-usable, and certainly not necessary. The idea of a Smart (tm) baby monitor might seem great until someone uses it to monitor your home network, capture your bank details and steal your money. Is an Internet connection **really** necessary for a baby monitor? Is it even a convenience? Or just a gimmick?

Smart energy meters are a hot topic in the UK and elsewhere at the moment, and there is some suggestion that having a Smart Meter will become mandatory at some point (it's not, in the UK, at the time of writing). There are some undeniable conveniences for bill-payers, although the conveniences for the providers seem to me to be more compelling. That's a distraction from a much more important issue, though: how much can we trust their security as an online device? Some people suggest that being able to inspect the source code running on devices such as meters would help.

And so we come back to Simplicity and the Software Crisis. Even if we are allowed to see the source code running on our IoT devices, that won't help if it's too complex to understand.

STEVE LOVE
FEATURES EDITOR

The official magazine of ACCU

ACCU is an organisation of programmers who care about professionalism in programming. That is, we care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

ACCU exists for programmers at all levels of experience, from students and trainees to experienced developers. As well as publishing magazines, we run a respected annual developers' conference, and provide targeted mentored developer projects.

The articles in this magazine have all been written by programmers, for programmers – and have been contributed free of charge.

To find out more about ACCU's activities, or to join the organisation and subscribe to this magazine, go to www.accu.org.

Membership costs are very low as this is a non-profit organisation.

DIALOGUE

REGULARS

FEATURES

SUBMISSION DATES

C Vu 29.2: 1st April 2017
C Vu 29.3: 1st June 2017

Overload 139: 1st May 2017
Overload 140: 1st July 2017

WRITE FOR C VU

Both C Vu and Overload rely on articles submitted by you, the readers. We need articles at all levels of software development experience. What are you working on right now? Let us know!

Send articles to cvu@accu.org. The friendly magazine production team is on hand if you need help or have any queries.

ADVERTISE WITH US

The ACCU magazines represent an effective, targeted advertising channel. 80% of our readers make purchasing decisions or recommend products for their organisations.

To advertise in the pages of C Vu or Overload, contact the advertising officer at ads@accu.org.

Our advertising rates are very reasonable, and we offer advertising discounts for corporate members.

COPYRIGHTS AND TRADE MARKS

On the Defensive

Pete Goodliffe demonstrates defensive programming techniques for robust code.

We have to distrust each other. It's our only defense against betrayal.
~ Tennessee Williams

It seems an age ago now. When my daughter was only 10 months old, she liked playing with wooden bricks. Well, she liked playing with wooden bricks and *me*. I'd build a tower as high as I could, and then with a gentle nudge of the bottom brick, she'd topple the whole thing and let out a little whoop of delight. I didn't build these towers for their strength – it would have been pointless if I did. If I had really wanted a sturdy tower, then I'd have built it in a very different way. I'd have shorn up a foundation and started with a wide base, rather than just quickly stacking blocks upon each other and building as high as possible.

Too many programmers write their code like flimsy towers of bricks; a gentle unexpected prod to the base and the whole thing falls over. Code builds up in layers, and we need to use techniques that ensure that each layer is sound so that we can build upon it.

Towards good code

There is a huge difference between code that *seems* to work, *correct* code, and *good* code. M. A. Jackson wrote, "The beginning of wisdom for a software engineer is to recognize the difference between getting a program to work, and getting it *right*." There *is* a difference:

- It is easy to write code that *works* most of the time. You feed it the usual set of inputs, it gives the usual set of outputs. But give it something surprising, and it might just fall over.
- *Correct* code won't fall over. For all possible sets of input, the output will be correct. But usually the set of all possible inputs is ridiculously large and hard to test.
- However, not all correct code is *good* code – the logic may be hard to follow, the code may be contrived, and it may be practically impossible to maintain.

By these definitions, good code is what we should aim for. It is robust, efficient enough and, of course, correct. Industrial strength code will not crash or produce incorrect results when given unusual inputs. It will also satisfy all other requirements, including thread safety, timing constraints, and re-entrancy.

It's one thing to write this good code in the comfort of your own home, a carefully controlled environment. It's an entirely different prospect to do so in the heat of the software factory, where the world is changing around you, the codebase is rapidly evolving, and you're constantly being faced with grotesque *legacy code* – archaic programs written by code monkeys that are now long gone. Try writing good code when the world is conspiring to stop you!

In this torturous environment, how do you ensure that your code is industrial strength? *Defensive programming* helps.

While there are many ways to construct code (object-orientated approaches, component based models, structured design, Extreme Programming, etc.), defensive programming is an approach that can be applied universally. It's not so much a formal methodology as an informal set of basic guidelines. Defensive programming is not a magical cure-all, but a practical way to prevent a pile of potential coding problems.

Assume the worst

When you write code, it's all too easy to make a set of assumptions about how it should run, how it will be called, what the valid inputs are, and so on. You won't even realize that you've assumed anything, because it all seems obvious to you. You'll spend months happily crafting code, as these assumptions fade and distort in your mind.

Or you might pick up some old code to make a vital last-minute fix when the product's going out the door in 10 minutes. With only enough time for a brief glance at its structure, you'll make assumptions about how the code works. There's no time to perform full literary criticism, and until you get a chance to prove the code is *actually* doing what you think it's doing, assumptions are all you have.

Assumptions cause us to write flawed software. It's easy to assume:

- The function won't *ever* be called like that. I will always be passed valid parameters only.
- This piece of code will *always* work; it will never generate an error.
- *No one* will ever try to access this variable if I document it *For internal use only*.

When we program defensively, we shouldn't make *any* assumptions. We should never assume that *it can't happen*. We should never assume that the world works as we'd expect it to work.

Experience tells us that the only thing you *can* be certain about is this: Your code will somehow, someday, go wrong. Someone *will* do a dumb thing. Murphy's law puts it this way: "If it can be used incorrectly, it will." Listen to that man – he spoke from experience [1]. Defensive programming prevents these accidents by foreseeing them, or at least fore-guessing them – figuring out what might go wrong at each stage in the code, and guarding against it.

defensive programming is an approach that can be applied universally

Is this paranoid? Perhaps. But it doesn't hurt to be a *little* paranoid. In fact, it makes a lot of sense. As your code evolves, you will forget the original set of assumptions you made (and real code does evolve). Other programmers won't have any knowledge of the assumptions in your head, or else they will just make their own invalid assumptions about what your code can do. Software evolution exposes weaknesses, and code growth hides original simple assumptions. A little paranoia at the outset can make code a lot more robust in the long run.

> Assume nothing. Unwritten assumptions continually cause faults, particularly as code grows.

Add to this the fact that things neither you nor your users have any control over can go wrong: disks fill up, networks fail, and computers crash. Bad things happen. Remember, it's never actually your program that fails – the software always does what you told it to. The actual algorithms, or perhaps the client code, are what introduce faults into the system.

PETE GOODLIFFE

Pete Goodliffe is a programmer who never stays at the same place in the software food chain. He has a passion for curry and doesn't wear shoes. Pete can be contacted at pete@goodliffe.net or @petegoodliffe

As you write more code, and as you work through it faster and faster, the likelihood of making mistakes grows and grows. Without adequate time to verify each assumption, you can't write robust code. Unfortunately, on the programming front line, there's rarely any opportunity to slow down, take stock, and linger over a piece of code. The world is just moving too fast, and programmers need to keep up. Therefore, we should grasp every opportunity to reduce errors, and defensive practices are one of our main weapons.

What is defensive programming?

As the name suggests, defensive programming is careful, guarded programming. To construct reliable software, we design every component in the system so that it *protects* itself as much as possible. We smash unwritten assumptions by explicitly checking for them in the code. This is an attempt to prevent, or at least observe, when our code is called in a way that will exhibit incorrect behaviour.

Defensive programming enables us to detect minor problems early on, rather than get bitten by them later when they've escalated into major disasters. All too often, you'll see 'professional' developers rush out code without thinking. Tinker with the code–run it–crash. Tinker–run–crash. Tinker–run–crash.

They are continually tripped up by the incorrect assumptions that they never took the time to validate. Hardly a promotion for modern day software engineering, but it's happening all the time. Defensive programming helps us to write correct software from the start and move away from the *code-it, try-it, code-it, try-it...* cycle.

Okay, defensive programming won't remove program failures altogether. Far from it. But problems will become less of a hassle and easier to fix. Defensive programmers catch falling snowflakes rather than drown under an avalanche of errors.

Defensive programming is a method of prevention, rather than a form of cure. Compare this to debugging – the act of removing bugs *after* they've bitten. Debugging is all about finding a cure.

Is defensive programming really worth the hassle? There are arguments for and against.

The case *against*:

- Defensive programming consumes resources, both yours and the computer's.
- It eats into the efficiency of your code; even a little extra code requires a little extra execution. For a single function or class, this might not matter, but when you have a system made up of 100,000 functions, you may have more of a problem.
- Each defensive practice requires some extra work. Why should you follow any of them? You have enough to do already, right? Well, then just make sure people use your code correctly. If they don't, then any problems are their own fault.

The case *for*:

- Defensive programming saves you literally hours of debugging and lets you do more fun stuff instead. Remember Murphy: If your code *can* be used incorrectly, it will be.
- Working code that runs properly, but ever-so-slightly slower, is *far* superior to code that works most of the time but occasionally collapses in a shower of brightly coloured sparks.
- We can design some defensive code to be physically removed in release builds, circumventing the performance issue. The majority of the items we'll consider here don't have any significant overhead, anyway.
- Defensive programming avoids a large number of security problems – a serious issue in modern software development. More on this below.

As the market demands software that's built faster and cheaper, we need to focus on techniques that deliver results. Don't skip the bit of extra work up-front that will prevent a whole world of pain and delay later.

What Defensive Programming Isn't

There are a few common misconceptions about defensive programming. Defensive programming is not:

Error checking

If there are error conditions that might arise in your code, you should be checking for them anyway. This is *not* defensive code. It's just plain good practice – a part of writing *correct* code.

Testing

Testing your code is not defensive. It's another normal part of our development work. Test harnesses aren't defensive; they can prove the code is correct now, but won't prove that it will stand up to future modification. Even with the best test suite in the world, anyone can make a change and slip it past untested.

Debugging

You might add some defensive code during a spell of debugging, but debugging is something you do after your program has failed. Defensive programming is something you do to *prevent* your program from failing in the first place (or to detect failures early before they manifest in incomprehensible ways, demanding all-night debugging sessions).

The big bad world

Someone once said, "Never ascribe to malice that which is adequately explained by stupidity." [2] Most of the time we are defending against stupidity, against invalid and unchecked assumptions. However, there *are* malicious users, and they will try to bend and break your code to suit their vicious purposes.

Defensive programming helps with program security, guarding against this kind of wilful misuse. Crackers and virus writers routinely exploit sloppy code to gain control of an application and then weave whatever wicked schemes they desire. This is a serious threat in the modern world of software development; it has huge implications in terms of the loss of productivity, money, and privacy.

Software abusers range from the opportunistic user exploiting a small program quirk to the hardcore cracker who spends his time deliberately trying to gain illicit access to your systems. Too many unwitting programmers leave gaping holes for these people to walk through. With the rise of the networked computer, the consequences of sloppiness become more and more significant.

Many large development corporations are finally waking up to this threat and are beginning to take the problem seriously, investing time and resources into serious defensive code work. In reality, it's hard to graft in defences *after* an attack.

Techniques for defensive programming

So what does all this mean to programmers working in the software factory?

There are a number of common sense rules under the defensive programming umbrella. People usually think of *assertions* when they think of defensive programming, and rightly so. We'll talk about those later. But there's also a pile of simple programming habits that will immeasurably improve the safety of your code.

Despite seeming common sense, these rules are often ignored – hence the low standard of most software at large in the world. Tighter security and reliable development can be achieved surprisingly easily, as long as programmers are alert and well informed.

The next few sections list the rules of defensive programming. We'll start off by painting with broad strokes, looking at high-level defensive techniques, processes, and procedures. As we progress, we'll fill in finer detail, looking more deeply at individual code statements. Some of these defensive techniques are language-specific. This is natural – you have to put on bulletproof shoes if your language lets you shoot yourself in the foot.

As you read this list, evaluate yourself. How many of these rules do you currently follow? Which ones will you now adopt?

Employ a good coding style and sound design

We can prevent most coding mistakes by adopting a good coding style. Simple things like choosing meaningful variable names and using parentheses judiciously can increase clarity and reduce the likelihood of faults slipping past unnoticed.

Similarly, considering the larger scale design before ploughing into the code is key. "The best documentation of a computer program is a clean structure." [3]. Starting off with a set of clear APIs to implement, a logical system structure, and well-defined component roles and responsibilities will avoid headaches further down the line.

Don't code in a hurry

It's all too common to see hit-and-run programming. Programmers quickly hack out a function, shove it through the compiler to check syntax, run it once to see if it works, and then move on to the next task. This approach is fraught with peril.

Instead, think about each line as you write it. What errors could arise? Have you considered every logical twist that might occur? Slow, methodical programming seems mundane – but it really does cut down on the number of faults introduced.

> More haste, less speed. Always think carefully about what you're typing *as* you type it.

A particular C-family gotcha that snares speedy programmers is mistyping `==` as just `=`. The former is a test for equality, the latter a variable assignment. With an unhelpful compiler (or with warnings switched off) there will be no indication that the program behaviour is not what was intended.

Always do *all* of the tasks involved in completing a code section before rushing on. For example, if you decide to write the main flow first and the error checking/handling second, you must be sure you have the discipline to do both. Be very wary of deferring the error checking and moving straight on to the main flow of three more code sections. Your intention to return later may be sincere, but later can easily become much later, by which time you will have forgotten much of the context, making it take longer and be more of a chore. (And of course, by then there will be some artificially urgent deadline.)

Discipline is a habit that needs to be learned and reinforced. Every time you don't do the right thing now, you become more likely to continue not doing the right thing in the future. Do it now, don't leave it for a rainy day in the Sahara. Doing it later actually requires *more* discipline than doing it now!

Trust no-one

Your mother told you never to talk to strangers. Unfortunately, good software development requires even more cynicism and less faith in human nature. Even well-intentioned code users could cause problems in your program; being defensive means you can't trust anybody.

You might suffer problems because of:

- *Genuine users* accidentally giving bogus input or operating the program incorrectly.
- *Malicious users* trying to consciously provoke bad program behaviour.
- *Client code* calling your function with the wrong parameters or supplying inconsistent input.
- *The operating environment* failing to provide adequate service to the program.

Always do ALL of the tasks involved in completing a code section before rushing on

Say "when"

When do you program defensively? Do you start when things go wrong? Or when you pick up some code you don't understand?

No, these defensive programming techniques should be used *all the time*. They should be second nature. Mature programmers have learned from experience – they've been bitten enough times that they know to put sensible safeguards in place.

Defensive strategies are much easier to apply as you start writing code, rather than retrofitting them into existent code. You can't be thorough and accurate if you try to shoehorn in this stuff late in the day. If you start adding defensive code once something has gone wrong, you are essentially debugging – being reactive, not preventative and proactive.

However, during the course of debugging, or even when adding new functionality, you'll discover conditions that you'd like to verify. It's always a good time to add defensive code.

- *External libraries* behaving badly and failing to honour interface contracts that you rely on.
- *You* might even make a silly coding mistake in one function or forget how some three-year-old code is supposed to work and then use it badly.

Don't assume that all will go well or that all code will operate correctly. Put safety checks in place throughout your work. Constantly watch for weak spots, and guard against them with extra defensive code.

> Trust no one. Absolutely anyone – including yourself – can introduce flaws into your program logic. Treat all inputs and all results with suspicion until you can prove that they are valid.

Write code for clarity, not brevity

Whenever you can choose between concise (but potentially confusing) code and clear (but potentially tedious) code, use code that *reads* as intended, even if it's less elegant. For example, split complex arithmetic operations into a series of separate statements to make the logic clearer.

Think about who might read your code. It might require maintenance work by a junior coder, and if he can't understand the logic, then he's bound to make mistakes. Complicated constructs or unusual language tricks might prove your encyclopedic knowledge of operator precedence, but it really butchers code maintainability. *Keep it simple.*

If it can't be maintained, your code is not safe. In really extreme cases, overly complex expressions can cause the compiler to generate incorrect code – many compiler optimization errors come to light this way.

> Simplicity is a virtue. Never make code more complex than necessary.

Don't let anyone tinker with stuff they shouldn't

Things that are internal should stay on the inside. Things that are private should be kept under lock and key. Don't display your code's dirty laundry in public. No matter how politely you ask, people *will* fiddle with your data when you're not looking if given half a chance, and they *will* try to call 'implementation only' routines for their own reasons. Don't let them.

- In object-oriented languages, prevent access to internal class data by making it private. In C++, consider the Cheshire cat/pimpl idiom.
- In procedural languages, you can still employ object-oriented (OO) packaging concepts, by wrapping private data behind opaque types and providing well-defined public operations on them.
- Keep all variables in the tightest scope necessary; don't declare variables globally when you don't have to. Don't put them at file scope when they can be function-local. Don't place them at function scope when they can be loop-local.

Compile with all warnings switched on

Most languages' compilers draw on a vast selection of error messages when you hurt their feelings. They will also spit out various *warnings* when they encounter potentially flawed code, like the use of a C or C++ variable before its assignment [4]. These warnings can usually be selectively enabled and disabled.

If your code is full of dangerous constructs, you'll get pages and pages of warnings. Sadly, the common responses are to disable compiler warnings or just ignore the messages. Don't do either.

Always enable your compiler's warnings. And if your code generates any warnings, fix the code immediately to silence the compiler's screams. Never be satisfied with code that doesn't compile quietly when warnings are enabled. The warnings are there for a reason. Even if there's a particular warning you think doesn't matter, don't leave it in, or one day it will obscure one that *does* matter.

> Compiler warnings catch many silly coding errors. Always enable them. Make sure your code compiles silently.

Use static analysis tools

Compiler warnings are the result of a limited *static analysis* of your code, a code inspection performed *before* the program is run.

There are many separate static analysis tools available, like *lint* (and its more modern derivatives) for C and FxCop for .NET assemblies. Your daily programming routine should include use of these tools to check your code. They will pick up many more errors than your compiler alone.

if your code generates any warnings, fix the code immediately to silence the compiler's screams

Use safe data structures

Or failing that, use dangerous data structures safely.

Perhaps the most common security vulnerability results from *buffer overrun*. This is triggered by the careless use of fixed-size data structures. If your code writes into a buffer without checking its size first, then there is always potential for writing past the end of the buffer.

It's frighteningly easy to do, as this small snippet of C code demonstrates:

```
char *unsafe_copy(const char *source)
{
  char * buffer = new char[10];
  strcpy(buffer, source);
  return buffer;
}
```

If the length of the data in **source** is greater than 10 characters, its copy will extend beyond the end of **buffer**'s reserved memory. Then anything could happen. In the best case, the result would be data corruption – some other data structure's contents will be overwritten. In the worst case, a malicious user could exploit this simple error to put executable code on the program stack and use it to run his own arbitrary program, effectively hijacking the computer. These kinds of flaw are regularly exploited by system crackers – serious stuff.

It's easy to avoid being bitten by these vulnerabilities: don't write such bad code! Use safer data structures that don't allow you to corrupt the program – use a managed buffer like C++'s **string** class. Or systematically use safe operations on unsafe data types. The C code above can be secured by swapping **strcpy** for **strncpy**, a size limited string copy operation:

```
char *safer_copy(const char *source)
{
  char * buffer = new char[10];
  strncpy_(buffer, source, 10);
  return buffer;
}
```

Check EVERY return value

If a function returns a value, it does so for a reason. Check that return value. If it is an error code, you *must* inspect it and handle any failure. Don't let errors silently invade your program; swallowing an error can lead to unpredictable behaviour.

This applies to user-defined functions as well as standard library ones. Most of the insidious bugs you'll find arise when a programmer fails to check a return value. Don't forget that some functions may return errors through a different mechanism (i.e., the standard C library's **errno**). Always catch and handle appropriate exceptions at the appropriate level.

Handle memory (and other precious resources) carefully

Be thorough and release any resource that you acquire during execution. Memory is the example of this cited most often, but it is not the only one. Files and thread locks are other precious resources that we must use carefully. Be a good steward.

Don't neglect to close files or release memory because you think that the OS will clean up your program when it exits. You really don't know how long your code will be left running, eating up all file handles or consuming all the memory. You can't even be sure that the OS will cleanly release your resources – some OSs don't.

There is a school of thought that says, "Don't worry about freeing memory until you know your program works in the first place; only then add all the relevant releases." Just say no. This is a ludicrously dangerous practice. It will lead to many, many errors in your memory usage; you *will* inevitably forget to free memory in some places.

> Treat all scarce resources with respect. Manage their acquisition and release carefully.

Java and .NET employ a *garbage collector* to do all this tedious tiding up for you, so you can just 'forget' about freeing resources. Let them drop to the floor, since the runtime sweeps up every now and then. It's a nice luxury, but don't be lulled into a false sense of security. You still have to think. You have to explicitly drop references to objects you no longer care about or they won't be cleaned up; don't accidentally hold on to an object reference. Less advanced garbage collectors are also easily fooled by circular references (e.g., *A* refers to *B*, and *B* refers to *A*, but no one else cares about them). This could cause objects to never be swept up; a subtle form of memory leak.

Initialize all variables at their points of declaration

This is a clarity issue. The intent of each variable is explicit if you initialize it. It's not safe to rely on rules of thumb like: *If I don't initialize it, I don't care about the initial value*. The code will evolve. The uninitialized value may turn into a problem further down the line.

C and C++ compound this issue. If you accidentally use a variable without having initialized it, you'll get different results each time your program runs, depending on what garbage was in memory at the time. Declaring a variable in one place, assigning it later on, and then using it even later opens up a window for errors. If the assignment is ever skipped, you'll spend ages hunting down random behaviour. Close the window by initializing every variable as you declare it; even if the value's wrong, the behaviour will at least be predictably wrong.

Safer languages (like Java and C#) sidestep this pitfall by defining an initial value for all variables. It's still good practice to initialize a variable as you declare it, which improve code clarity.

Declare variables as late as possible

By doing this, you place the variable as close as possible to its use, preventing it from confusing other parts of the code. It also clarifies the

code using the variable. You don't have to hunt around to find the variable's type and initialization; a nearby declaration makes it obvious.

Don't reuse the same temporary variable in a number of places, even if each use is in a logically separate area. It makes later reworking of the code awfully complicated. Create a new variable each time – the compiler will sort out any efficiency concerns.

Use standard language facilities

C and C++ are nightmares in this respect. They suffer from many different revisions of their specifications, with more obscure cases left as implementation-specific *undefined behaviour*. Today there are many compilers, each with subtly different behaviour. They are mostly compatible, but there is still plenty of rope to hang yourself with.

Clearly define which language version you are using. Unless mandated by your project (and there had better be a good reason), *don't* rely on compiler weirdness or any non-standard extensions to the language. If there is an area of the language that is undefined, don't rely on the behaviour of your particular compiler (e.g., don't rely on your C compiler treating `char` as a `signed` value – others won't). Doing so leads to very brittle code. What happens when you update the compiler? What happens when a new programmer joins the team who doesn't understand the extensions? Relying on a particular compiler's odd behaviour leads to *really* subtle bugs later in life.

Use a good diagnostic logging facility

When you write some new code, you'll often include a lot of diagnostics to check what's going on. Should these really be removed after the event? Leaving them in will make life easier when you have to revisit the code, especially if they can be selectively disabled in the meantime.

There are a number of diagnostic logging systems available to facilitate this. Many can be used in such a way that diagnostics have no overhead if not needed; they can be conditionally compiled out.

Cast carefully

Most languages allow you to *cast* (or convert) data from one type to another. This operation is sometimes more successful than others. If you try to convert a 64-bit integer into a smaller 8-bit data type, what will happen to the other 56 bits? Your execution environment might suddenly throw an exception or silently degrade your data's integrity. Many programmers don't think about this kind of thing, and so their programs behave in unnatural ways.

If you really want to use a cast, think carefully about it. What you're saying to the compiler is, "Forget your type checking: *I* know what this variable is, you don't." You're ripping a big hole into the type system and walking straight through it. It's unstable ground; if you make any kind of mistake, the compiler will just sit there quietly and mutter, "I told you so," under its breath. If you're lucky (e.g. using Java or C#) the runtime might throw an exception to let you know, but this depends on exactly what you're trying to convert.

C and C++ are particularly vague about the precision of data types, so don't make assumptions about data type interchangeability. Don't presume that `int` and `long` are the same size and can be assigned to one another, even if you can get away with it on *your* platform. Code migrates platforms, but bad code migrates badly.

Conclusion

It is important to craft code that is not just correct but is also good. It needs to document all the assumptions made. This will make it easier to maintain, and it will harbour fewer bugs. Defensive programming is a method of expecting the worst and being prepared for it. It's a technique that prevents simple faults from becoming elusive bugs.

The use of codified constraints alongside defensive code will make your software far more robust. Like many other good coding practices (*unit testing*, for example), defensive programming is about spending a little extra time wisely (and early) in order to save much more time, effort, and cost later. Believe me, this *can* save an entire project from ruin. ■

Offensive programming?

Attack is the best form of defence.
~ Proverb

While writing this article, I wondered, *What's the opposite of defensive programming?* It's *offensive programming*, of course!

There are a number of people I know who you could call offensive programmers. But I think there's more to this than swearing at your computer and never taking baths.

It stands to reason that an offensive programming approach would be actively trying to *break* things in the code, rather than defending against problems. That is, actively attacking the code rather than securing it. I'd call that *testing*. Testing, when done properly, has an incredibly positive effect on your software construction. It improves code quality greatly and brings stability to the development process.

We should be all offensive programmers.

Questions

- Can you have *too much* defensive programming?
- Should you add an assertion to your code for every bug you find and fix?
- Should assertions conditionally compile away to nothing in production builds? If not, which assertions should remain in release builds?
- Are exceptions a better form of defensive barrier than C-style assertions?
- How carefully do you consider each statement that you type? Do you relentlessly check every function return code, even if you're *sure* a function will not return an error?

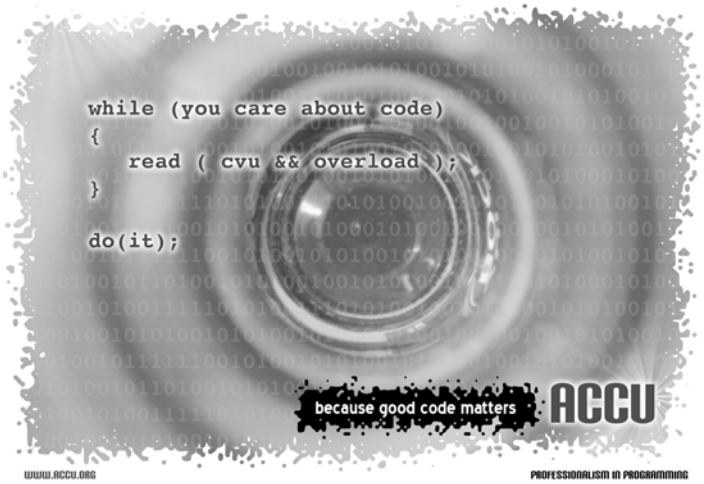
Notes and references

[1] Edward Murphy, Jr., was a US Air Force Engineer. He coined this infamous law after discovering a technician had systematically connected a whole row of devices upside down. Symmetric connectors permitted this avoidable mistake; afterwards, he chose a different connector design.

[2] Some historians attribute that quote to Napoleon Bonaparte. Now there's a guy who knew something about defence.

[3] *The Elements of Programming Style*. B.W. Kernighan, P.J. Plauger. 1978.

[4] Many languages (like Java and C#) class this as an error.



Beyond Functional Programming: Manipulate Functions with the J Language

Adam Tornhill explores a different kind of programming language.

The Pragmatic Programmer [1] recommends that we learn at least one new language every year. To be effective, the languages we learn should differ sufficiently from those we already master and ideally introduce us to a new paradigm too. Learning a different programming language affects the way we view code. A new paradigm may even alter our problem solving abilities by reshaping the way we think. The J programming language offers both of these qualities.

In this article, we'll get a brief introduction to the J programming language [2]. The goal is to whet your appetite for an exciting programming paradigm rather than providing an in-depth introduction. As we move along, we'll learn just enough J to explore a fascinating corner of the language that lets us manipulate functions as data. You'll see how these techniques differ from traditional functional programming, why manipulating functions is interesting and how it lets you solve real problems with a minimum of code.

The brevity of array languages

The J language stems from the family of array languages. Array languages sit on an interesting, albeit relatively unknown, branch in the family tree of programming languages. APL, the first array language, presents a radical departure from how we typically think about computer programming. Even though modern languages let us express our programs on a fairly high level, we're still basically moving and processing data element by element. This level of abstraction is a constant source of irregularity and special cases in code. APL – and more recent array languages like J – give us as an alternative and interesting contrast to the von Neumann languages of today.

Array languages sit on an interesting, albeit relatively unknown, branch in the family tree of programming languages

So what makes array languages so different? First of all, they're extremely compact. Let me show you how compact by exploring a piece of APL code: `x[⍋x←6?40]`. Alright. What was that? A burst of line noise that made its way past the editors into the article you're reading? No, no. What we just saw was a *complete* APL program. This brief program generates six lottery numbers in the range 1 to 40, no duplicates, and delivers them sorted in ascending order.

The first thing that strikes us in this APL example is the character set. APL uses a special set of symbols in its syntax. Today we can play around with virtual keyboards (see Try APL [3]), but in the 1960s, when APL was first implemented, the language required special hardware. It's probably fair to assume that requiring a curious programmer to buy a special keyboard and graphics card just to try a new programming language could be something of a deal breaker in terms of language adoption. That said,

ADAM TORNHILL

Adam is a programmer who combines degrees in engineering and psychology. He's the author of *Your Code as a Crime Scene*, has written the popular 'Lisp for the Web' tutorial and self-published a book on *Patterns in C*. Adam also writes open-source software in a variety of programming languages.

I've become a fan of the APL syntax. I even think that special symbols and character sets are underused in today's programming languages. Symbols help us build compact mental models and make it easier to spot patterns and idioms in our code. It's a learning curve for sure, but a flat one.

When I started to talk about APL in some of my presentations (for example 'Code That Fits Your Brain' [4]), people often remark that APL looks opaque and cryptic. If you agree with that statement, you'd be happy to learn that the J language abandoned APL's symbols for a pure ASCII syntax. So let's translate our APL example above to J and see if we can make sense of it: `` `/:~>:6?40` ``. Much simpler, right? It's 100% ASCII after all.

As evident from this example, J is also a compact language. The example also shows that a different representation, like ASCII, doesn't necessarily improve our understanding in the strange world of array languages. That's because this is not a matter of syntax; array languages present a radically different way to code and solve problems. Let's explore how.

Meet true simplicity in the J language

What fascinates me the most about our APL and J examples is not so much what's there, but rather what's *absent*. Have a look at the code once again. There is no conditional logic. Even more interesting, there are no explicit loops even though the code models an inherently iterative problem.

The reason array languages pull this off is because arrays are the fundamental datatype. That means functions in J operate on collections of things rather than individual values. The implication is that as soon as we start to think about data as collections rather than individual elements, a whole set of potential coding issues just go away. Suddenly there's no difference if we have 1,000, a single value or none; our J code will look exactly the same. That means we can get rid of many special cases that we need to deal with in other languages. For example, think of the disastrous 'null' pointers that plague most mainstream languages. In J, you'd just model the absence of something as an empty array; the rest of your code stays the same. It's a simple yet powerful abstraction that lets you focus on the problem you're trying to solve without littering your code with special cases.

Alright, let's move ahead and get started with some J. I'd suggest that you download the latest version of the J language and environment [5] to follow along as we walk though some examples.

Our first example is fairly trivial:

```
   1 + 1
2
```

In the example above we add two numbers and J, since it's an interactive and interpreted language, prints the result immediately. As you see, the code you write yourself (like `1 + 1`) is always indented in the J prompt. Results and errors from the J runtime are left aligned.

Next we'll do something more interesting. Let's repeat our addition, but this time with two arrays:

```
   1 2 3 + 4 5 6
5 7 9
```

Remember when I said that arrays are fundamental data types in J? That's the array capabilities I've been talking about. We could repeat this operation with data of higher dimensions stored in multi-dimensional arrays. Our code would look the same and the J interpreter would handle the looping for us. But J can do more. Let's see how we can calculate the sum of the numbers in an array:

```
   +/ 1 2 3 4 5
15
```

We've already met `+`, but what's that slash character we put after it? And how does J manage to sum our array without any explicit loop? I'll answer it in a minute, but first we need to learn a bit of J jargon. J doesn't really have functions. Instead, the language borrows its terminology from natural languages. `+` is a *verb* in J speech. And `/` is an *adverb* that, just as in natural languages, modifies its verb. In this case, the slash inserts `+`, the verb it refers to, between each number in our array. That is, `+/` has the same effect as writing `1 + 2 + 3 + 4 + 5`. If you're into functional programming, you can think of an adverb as a higher-order function; it takes a function as input and returns a new function that modifies the behavior of the original function.

The simplicity of J goes deeper than avoiding explicit loops. J's verbs have a remarkable simplicity in their definitions. It turns out that a verb/function in J always takes either one or two arguments and you usually don't even name those arguments. That's it. No more code review battles over the one true way to name an index argument. No need for syntactic sugar like keywords and variable arguments. Just one or two implicit arguments. Follow along and I'll explain how we program with those constraints by introducing a *tacit* style of programming.

Tacit programming

The J language supports a **tacit** programming style (also called *point-free style*). A tacit style means that our functions don't identify the arguments they operate upon. We can exemplify a tacit function by specifying a verb of our own:

```
   sum =: +/
```

The code above introduces a user defined verb names `sum`. As you see, there's no mention of any variable or function argument: the argument is implicit. We can now use `sum` like you'd expect:

```
   sum 1 2 3 4 5
15
```

Yeah – works just like our previous example with adverbs when we typed `+/ 1 2 3 4 5`. Our `sum` verb is called a *monadic definition*. The term is unrelated to the kind of monads you'd find in Haskell land. Instead *monadic* means that it's a function that takes *exactly* one argument. That argument is always given to the right of the verb as you see in the invocation of `sum` in the example above.

The other category of verbs in J are called `dyadic`. A dyadic takes exactly two arguments, one to its left and one to its right. A simple example is the addition `1 + 1` that we met earlier.

Learn to box

Limiting functions to only one or two arguments sounds like quite a constraint. The reason it works so well in J is because, remember, our arguments are always arrays and these arrays can be of any dimension. Learning to re-frame our understanding of a problem to be able to express it as an operation on arrays is part of the challenge when you start out with J. However, that learning is likely to translate to other languages too. After all, taking some input, doing something to it, and generating some output is pretty much all a computer program does. J helps us get better at expressing that universal pattern.

But of course, sometimes we do run into limitations in J too. For example, arrays in J have to contain the same data type. Let's say you want to pass an array with an integer and a string into one of your verbs. In that case you need a **box**. A box in J is a data type that supports heterogeneous values. Listing 1 shows how we work with boxes.

Listing 1

```
   b =: (< 1) , (< 'hello')
   b
+-------+
¦1¦hello¦
+-------+
   # b
2
```

In the first line we use the monadic verb `<` to put the value `1` into a box and the string `hello` into another box. We then use the dyadic verb `,` (called Append in J) to create an array out of our two boxes. The result of Append is assigned to our variable `b`. You also see that J has a nice printed representation of our boxed values. Finally we meet the monadic verb `#` (called Tally) that counts the number of items in our boxed array. Just as we'd expect, Tally reports two items.

Armed with our basic J knowledge, we're ready to get some serious business going by starting to manipulate verbs just as if they were data too.

Write functions that manipulate functions

The J language has several functional elements although it's not a strict functional language. However, it does take function abstraction a step further.

The basic idea in functional programming is that functions are first-class citizens. This promise looks rather shallow once we see its realization in different languages. Sure, we can pass functions as arguments, return functions from other functions and even compose functions. But there are few languages that allow us to *really* manipulate functions in the same way we massage data. For example, what would it mean to subtract one function from another? Or to calculate the derivate of a function? I'd say both of these examples make sense from a mathematical perspective, yet Clojure doesn't know how to do it and neither does Scala nor F#. This is where J takes off.

Again we'll start simple. The verb Double `+:` does what it says and doubles its input values:

```
   +: 2
4
   +: 3
6
```

We talked about J's terminology earlier and so far we've met verbs and adverbs. J also has *conjunctions*. In natural language grammar, a conjunction is something that connects different words and phrases. J's conjunctions have a similar effect and we'll see that as we explore the Power Conjunction `^:`.

The power conjunction applies a given verb *n* times. Here's how it works when applied to our Double verb:

```
   (+: ^: 2) 2
8
   (+: ^: 2) 3
12
```

The examples above instruct the power conjunction to apply its verb, Double (`+:`), twice. We could of course also write `+: ^: 3` and have the Double verb applied three times. Let's capture that in a user defined verb, tacit style:

```
   threeTimes =: +: ^: 3
   threeTimes 2
16
   threeTimes 1 2 3 4 5
8 16 24 32 40
```

As you see in the example above, our user defined verb `threeTimes` automatically generalizes to whole arrays. Instead of spending precious code on loops or iterations, we just let the language do the job for us. That

leaves our code free to focus on the stuff that actually does something, the core of the problem we're trying to solve.

There's a particular beauty to conjunctions. But J's way of manipulating functions go deeper. We can let the language **inverse** the meaning of a verb. As we'll see soon, inversing verbs is more than just a neat party trick. Once we have the ability of automatically inversing a function we are able to abstract several common programming tasks using succinct idiomatic expressions.

Inverse functions with the Power Conjunction

As I decided to learn the J language I looked for some small yet realistic problems to work on. I decided to start with the Dyalog APL challenge, an annual APL programming competition, but doing it in J instead. The 2014 competition [6] had a simple and interesting problem labelled 'How tweet it is'. The task here is to shorten a message, yet retain most of its readability, by removing interior vowels from words. For example, the phrase 'APL is REALLY cool' would be shortened to 'APL is RLLY cl'. Let's give it a try.

The first step in this task is to tokenize the string. That's straightforward with J's Words verb (written `;:`):

```
   b =: ;: 'APL is really cool'
   b
+------------------+
¦APL¦is¦really¦cool¦
+------------------+
```

The `;:` verb splits our string on its separators. It also partitions the resulting sub-string into a box since the tokens have different lengths. We let the variable `b` refer to our boxed array so that we can play around with it in the J interpreter and iterate towards our solution.

The next step towards a tweetable sentence is to remove the interior vowels. This is fairly straightforward, but the details would distract us from our main topic of function inverses as we would need to learn more J verbs. So instead, we'll pretend that we already have a verb `trimWord` that does the job for us (head over to my GitHub repository [7] if you'd like to look at its implementation).

To use our `trimWord` verb we need to take the words out of the box, apply `trimWord`, and put the results back into a boxed array. We've already seen how the `<` verb lets us box a value. J provides a corresponding unbox operation with the `>` verb. Semantically, unbox (`>`) is the inverse of box (`<`). As a programmer, we immediately see that. More interesting, the J language knows about it.

J has a unique language feature that lets you inverse the meaning of a verb. It's a mechanism that works on most built-in verbs and, fascinatingly, also on most of our own user defined verbs. And in the cases where J isn't able to deduce an inverse automatically, you can teach J about it by assigning an inverse yourself.

We already learned that the Power Conjunction `^:` lets us control how many times a verb is applied. So what happens if we apply the power conjunction -1 times? That's right – we get an inverse that undoes the effect of applying the particular verb. Let's play around with this idea in the J interpreter to see how it may solve our tweetable problem. I've introduced comments (`NB.`) to explain the steps. Please note that negative numbers are entered with an underscore, like `_1`, in the J interpreter (see Listing 2).

The example in Listing 2 shows a mathematical inverse like plus and minus. We also see a logical relationship between box and unbox (those kind of non-mathematical relationships are the reason J prefers the term 'obverse' over 'inverse').

So far I hope you find the inverse trick just as cool as I do. But how does it apply to our tweetable problem? Well, let's recap the pattern we want to express: first we'll unbox our words, then we let J apply `trimWord` to each of the words before we box the results. As we'll soon see, the general form of this pattern is common to many familiar programming problems. That's why J provides an idiomatic expression for it.

In J speech, this pattern is called Under (`&.`). Under works on two verbs that form a pipeline: the first verb is applied, then the second verb is applied to that result before the *inverse* of the first verb is applied to form the final result. Let's see it in action:

```
   b =: ;: 'APL is really cool'
      b
+------------------+
¦APL¦is¦really¦cool¦
+------------------+
   (trimWord&.>) b
+--------------+
¦APL¦is¦rlly¦cl¦
+--------------+
```

And there we are – we have a tweetable sentence. However, there's a bit more work before we're done. That's good because it gives us an opportunity to uncover some more J ideas.

Turn the dial to 11: Under under an Under

Our solution so far works as long as we have our words in an array of boxes. But our starting point is a raw string, not a box. How do we glue that together? Well, since we now have a solid grip on function inverses with Under, we'll use the same idiom. We tokenize the string with `;:`, trim the words in a box using our Under pipeline `trimWord&.>` before we put the trimmed tokens back into a string with the inverse to `;:`. Here are those steps in J code. Note how J automatically deduces the inverse to the tokenize (`;:`) verb:

```
   tweetable =: trimWord&.> &.;:
   tweetable 'APL is really cool'
APL is rlly cl
```

As you might note in our solution above, J executes expressions from right to left. Sure, as you dive deeper into J, you'll see that there are some exceptions. Or, to be more correct, J has a certain set of execution patterns

Listing 2

```
   addTwo =: 2 & +            NB. & binds one argument of a verb (think partial application)
   addTwo 1 2 3 4 5
3 4 5 6 7
   undoAddTwo =: addTwo ^: _1 NB. Create the inverse of addTwo.
     NB. That is, a verb that undoes the effect of addTwo
   undoAddTwo 3 4 5 6 7
1 2 3 4 5

   b =: < 1  NB. box the value 1
   b         NB. type the variable name to see its content
+-+
¦1¦
+-+
   unbox =: < ^: _1 NB. undo the effect of box by taking its inverse
   unbox b
1
```

that help your express more functionality with less code (these patterns are called 'forks' and 'hooks', which we won't cover this time).

Function inverses in the real world

Function inverses are the kind of construct that change how we view code. Once you've learned about function inverses, you'll see use cases everywhere. It turns out to be a general pattern, no matter what programming language we use. For example, consider pairs of `malloc`/`free`, `acquire`/`release`, and `open`/`close`. All of them follow the pattern captured in the Under idiom: We do something to create a context, apply some functions in that context, and then leave our context by undoing the initial effects with a logical inverse like `free`, `close`, etc.

Most modern languages provide some mechanism to capture these steps – in C++ we use the RAII idiom, in Lisp we express it with macros, and in Java, well, we just wait a few years for an extension of the core language – but no approach is as succinct and powerful as J's.

Exploring the J language is likely to change how you view programming and make you think differently about both functions and data. The Under idiom introduced in this article is just one example. J is full of mind melting ideas. For instance, just as we can inverse a function we can also derivate it or provide a list of functions as a declarative alternative to traditional conditional logic. I hope to explore those paths in future articles.

How hard is J to learn?

If this article is your first exposure to J, you probably find the language tricky at best and pure line noise at worst. You're in good company; the Internets are full of attempts at learning to decipher array languages. My favourite is a blog post by Ron Jeffries [8] that starts with "J. This is HARD. However...No, really. This is hard". I owe a lot to Ron since it was his blog post that turned my attention to J. Somehow I thought I needed that challenge (it was a good day). I also found J hard to learn and I still struggle with problems I know how to solve in languages I'm more comfortable with.

However, if J makes it hard to write code, perhaps that's a good thing; We don't need more code. We need less code but with higher quality. The best way to achieve that is by re-framing and simplifying the problems we try to solve. J helps us with that.

That said, I don't think J is hard *per se*. J is only hard to learn if you already know how to program. In that case there are lots of habits to unlearn. We also need to break our automated interpretation of common programming symbols since they carry a radically different meaning in J. For example, a seasoned programmer expects curly braces and parentheses to be balanced. That's not the case in J where those symbols form different verbs. Breaking those code reading habits takes time, yet it's kind of the easy part. The hard part is to change how we approach problem solving in code.

Finally, if you struggle with the J language, you'll always find comfort in Dijkstra's words [9] about its predecessor: "APL is a mistake, carried through to perfection". Now we know that Dijkstra was slightly wrong; J clearly carried both that mistake and its perfection further. ■

References

[1] *The Pragmatic Programmer*, https://www.amazon.com/Pragmatic-Programmer-Journeyman-Master/dp/020161622X
[2] http://jsoftware.com/
[3] http://tryapl.org/
[4] https://www.youtube.com/watch?v=NoGn2RWbBoE
[5] http://jsoftware.com/stable.htm
[6] http://www.dyalog.com/uploads/files/student_competition/2014_problems_phase1.pdf
[7] https://github.com/adamtornhill/apl-challenge-2014-in-J
[8] http://ronjeffries.com/xprog/articles/j-this-is-hard-however/
[9] http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD498.html

Be Available, Not Busy

Chris Oldwood considers how agility is best achieved.

We all like to be busy; at least most people I've worked with would much prefer to get on with something than just sit around all day twiddling their thumbs. That's not to say taking a break once in a while is not desirable, or even beneficial, but that by-and-large we generally have a good work ethic which means we want to pull our weight and make an equal contribution to the team's efforts. In essence, if we're always busy, we can't be accused of not earning our keep through being idle.

The best-laid plans…

Anyone who's ever had to employ builders to do any non-trivial building work, like an extension for example, will be all too aware of how detrimental busy people can be to a project's progress. The extension starts great – the builders are in full time so the footings and walls spring up nicely and they move on to the roof. Then the unexpected happens – it starts to rain – and the builders don't appear that day. It might be for safety reasons they haven't shown up, or just because no one likes working outside in the cold, driving rain when they could be doing something else in the warm indoors instead.

The following day, it's raining again and it's a no-show as before, fair enough. But by the fourth day the rain has stopped and the builders are still nowhere to be seen, so you give them a call to find out what's going on. It turns out it was unsafe for them to put on the roof in the horrendous weather so they spent the last few days on another job whilst waiting for the weather to clear up. Despite the rain stopping they decided to finish up this other job as it was only going to be a couple of days. Only, this other job hasn't quite gone according to plan either and the owner currently has no windows so they really need to get that sorted before coming back to carry on your extension. They've sent the faulty ones back and the manufacturer has absolutely promised them replacements this afternoon…

Different strokes

And so it goes on. This is a classic example of how people's desire to be busy means that their own agenda comes before that of the project – in this instance, your extension. Your goal is probably to have the extension built as quickly and cheaply as possible with the minimum of disruption, whereas the builder's is likely after maximising their income. To offer the cheapest price and yet still earn income on days where work on your project is not possible, they will find other paid work to do. The alternative for you and them would probably be a retainer in the form of a more expensive overall cost. Don't get me wrong, I'm not suggesting that builders do not have our interests at heart, as I'm sure they want a successful outcome too, but that their top priority and the customer's top priority may not always be in alignment.

This effect of context switching and delay often becomes even more apparent as the project moves to completion. Once the structural work is done and we move onto the plumbing, electrics and decorating then, every time we have a handover or need a different trade, we may have to wait an indeterminate amount of time for them to become free if they are not being very tightly orchestrated. If the job is not done satisfactorily, any rework may also have to be fitted in around other client's jobs and so the eventual date of completion gradually slips back again, and again.

So that's how it pans out in small scale construction, but how about for software delivery?

CHRIS OLDWOOD

Chris is a freelance programmer who started out as a bedroom coder in the 80's writing assembler on 8-bit micros. These days it's enterprise grade technology in plush corporate offices. He also commentates on the Godmanchester duck race and can be easily distracted via gort@cix.co.uk or @chrisoldwood

Virtual extensions

There are potentially different forces at play here because of inherent differences in the two industries, but there are definitely parallels, even when you have a dedicated team on a software project. Unless the team is truly cross-functional with the full gamut of infrastructure and security skills, along with complete autonomy, it will require collaboration with people outside their control. The question is whether inside the team, at least, they can operate in a way that reduces delays and maximises the flow of features.

lower priority work can starve the more important stuff when it can't be interrupted

Intuition tells us that if everyone is working as efficiently as possible then the whole team must, by definition, also be working as efficiently as possible. Hence if we're always busy we are being efficient (time wise) because there is no slack in the day. And this all makes perfect sense if the aim is to keep people busy, but sadly it does not appear to be the most effective way to get new features and fixes as quickly into production as possible.

Dampening inertia

There is a moment in Eliyahu Goldratt's *The Goal* where there is a suggestion that it may be more beneficial for a worker to be idle for 10 minutes than to immediately get on with the next piece of work. This almost certainly sounds like heresy and goes against our intuition, which is likely why the characters in the book were suspicious of the hypothesis too. The difference of course is that the work they could pick up now might not be as important as the piece they pick up in 10 minutes time. If they start work on a lower priority task and have to finish it, it will delay the start of the higher priority one. Anyone who's ever looked into priority inversion problems in a multi-threaded system will be all too aware of how lower priority work can starve the more important stuff when it can't be interrupted. The overall effect is that work *is* being done but it's not necessarily the work we *want* to be done right *now*.

In software delivery, we're ideally looking to ensure that the moment a piece of work enters the team, the finished article pops out the other end as quickly as possible. The quicker we put a feature into production the quicker we can exploit its 'value'. Potentially more importantly, though, is that the faster we can move a change through the pipeline, the quicker we can remediate unforeseen problems and react to new customer feedback. Unless you've got a product owner who paradoxically believes 'everything is the number one priority' then you will find greater riches in ensuring you finish what you start, rather than keep starting new things.

Small is beautiful

Context switching, as we know from watching computers thrash about when they have too many tasks on the go at once, does not appear to be the answer – smaller units of change (done to completion), however,

appears to work better. If you're used to working on features that take many days or weeks to complete, then it might seem impossible to imagine how you can break your work down into such small pieces that they generally only last a few hours to a couple of days, but in many cases you can.

For example, refactoring work, by definition, does not change the observable behaviour of the system and therefore is a candidate for delivery into production the moment it's complete. Similarly, the use of feature toggles enables us to deliver working code straight into production, even if real users don't have access to it whilst we iron out the wrinkles. Finally the use of 'spikes' allows us to decouple the act of learning about a problem from delivering the solution to it.

Outside of writing production code, the modern programmer has many other tasks on their plate, which can be separated off into distinct units of work. These can be independently prioritised and executed, such as documentation, updating tools, reviewing other people's changes, improving the deployment process, monitoring, etc. Even if all that is done and dusted there is always far more learning to do than we have hours in the day so reading a few blog posts, book chapters or articles about the tools and technology stack is another excellent way to spend a very small amount of time in a valuable way. Note that this should not be 'busy work' work invented just to keep you out of trouble; everything we do should be because it adds value to the product directly or indirectly through improving the delivery process – 'being available' does not mean 'killing time'.

The net result of this decomposition of chunks of work into much, much smaller units is that it's easier to smooth out the flow because at the huddle each day you can re-prioritise based on the previous day's events. With really small tasks (on most days) people in the team are available to either pick something new up or even pair or mob with others to help get the most important feature out the door. If someone goes off sick for a few days, rather than waiting for their return you could probably finish whatever it was they were doing, or even repeat their work again because the time lost is minimal and you may generate more disruption by trying to work around their unfinished efforts.

Being busy is a natural state that we will find ourselves in most of the time, but that should be because the work we are doing is effectively the most useful thing we could be doing at that moment in time. What we need to be aware of is the consequence of being busy *all* the time so that we begin to lose sight of the proverbial wood hidden behind the trees. Not all tasks are created equally and some even become redundant before we've started them, so it's important that we keep re-evaluating what we're doing and why. If you begin to realise you can fork off the latter part of a task into a separate lower value one and finish early, you will be available to trade-off lower value work for something higher. Whether you actually do or not is somewhat irrelevant, being able to is what really counts.

being available to work on the next most important thing helps ensure that the money is biased towards the more valuable pursuits

Culture shock

Switching management styles to one that focuses on the outcome of the work instead of what each 'resource' is up to is hugely liberating for the workers but also difficult for some to grasp. When a team self-organises almost on a daily basis to meet the demands of the product backlog it may be harder to know exactly what any one individual is up to or what contribution they may make to any single piece of work.

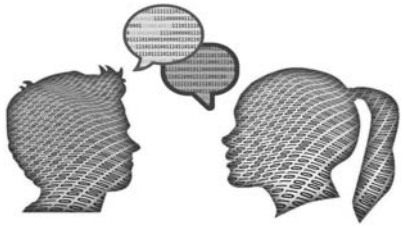
In essence, it shouldn't matter as long as the cost of developing any feature is less than the value it generates, but given how hard it is to put a figure on that for any one item, it's even more important that the team continuously delivers a working product to see how the costs are measuring up. Always being available to work on the next most important thing helps ensure that the money is biased towards the more valuable pursuits rather than being diluted within bigger tasks of mixed value.

Summary

Agility comes from being in a position to make a change in direction sooner. To react to the ever changing landscape, which could easily be a daily occurrence (or sometime less!) the members of the team should work in a way that allows them to start *and* complete 'useful' tasks in a small period of time. Being more aware of what is going on around us and where our current piece sits in the pecking order means we can make a judgement call on whether to soldier on alone, call for reinforcements or even stop short and offer ourselves up instead. If the spotlight switches from the people to the outcome then it helps those people to self-organise around that goal when their availability is higher. ■

Code Critique Competition 104

Set and collated by Roger Orr. A book prize is awarded for the best entry.



Please note that participation in this competition is open to all members, whether novice or expert. Readers are also encouraged to comment on published entries, and to supply their own possible code samples for the competition (in any common programming language) to scc@accu.org.

Note: If you would rather not have your critique visible online, please inform me. (Email addresses are not publicly visible.)

Last issue's code

I am trying to keep track of a set of people's scores at a game and print out the highest scores in order at the end: it seems to work most of the time but occasionally odd things happen...

Can you see what's wrong?

The code – scores.cpp – is in Listing 1.

Listing 1

```
#include <functional>
#include <iostream>
#include <map>
#include <sstream>
#include <unordered_map>

// Best scores
std::multimap<int, std::string, std::less<>>
best_scores;

// Map people to their best score so far
std::multimap<int, std::string>::iterator typedef
entry;
std::unordered_map<std::string, entry>
peoples_scores;
entry none;
void add_score(std::string name, int score)
{
  entry& current = peoples_scores[name];
  if (current != none)
  {
     if (score <= current->first)
     {
       return; // retain the best score
     }
     best_scores.erase(current);
  }
  current = best_scores.insert({score, name});
}
void print_scores()
{
   // top down
   for (auto it = best_scores.end();
        it-- != best_scores.begin(); )
   {
      std::cout << it->second << ": "
        << it->first << '\n';
   }
}
int main()
{
  for (;;)
  {
    std::cout << "Enter name and score: ";
    std::string lbufr;
    if (!std::getline(std::cin, lbufr)) break;
    std::string name;
    int score;
    std::istringstream(lbufr)
      >> name >> score;
    add_score(name, score);
  }
  std::cout << "\nBest scores\n";
  print_scores();
}
```

Listing 1 (cont'd)

Critique

Chris Main <cmain@fastmail.fm>

This is a tricky problem as I haven't been able to come up with any valid test data that makes 'odd things happen' on my platform, which is:

g++ 5.4 on Ubuntu 16.04, compiling with -std=c++11.

My first point is therefore the importance of a precise statement of the issue. Ideally this should state the actual input, the expected result and the actual result. Even if this is not possible because the issue cannot be repeatedly reproduced, the description should at least state the kind of issue, for example:

- the program crashes
- the program outputs spurious values
- the program outputs incorrect values
- the program outputs lines in the wrong order
- the program does not output as many lines as it should

Compilation failed for me because of the `std::less<>` in the declaration of `best_scores`. This can be fixed by changing it to `std::less<int>`, or better still by removing it altogether as that is the default. I assume this was just a typo.

There is no validation of input lines, so if something other than a name and a score is entered that might cause strange results, but the way the question is posed suggests that the problem is less obvious than that.

My first thought was iterator invalidation. Multimap iterators are being held as the `mapped_type` in an `unordered_map`. These iterators are associated with entries in `best_scores`, so I wondered whether they could become invalid when other entries are added to or removed from `best_scores`. However, after some research I found that multimap iterators are not invalidated in these cases.

In `add_score()`, the test for whether a person has a previous score is done by calling `operator[]` and comparing the result with a default constructed iterator. Although this works, it is more idiomatic to call

ROGER ORR

Roger has been programming for over 20 years, most recently in C++ and Java for various investment banks in Canary Wharf and the City. He joined ACCU in 1999 and the BSI C++ panel in 2002. He may be contacted at rogero@howzatt.demon.co.uk

`find()` and compare the result with the `end()` iterator. The function would then look like:

```
auto current = peoples_scores.find(name);
if (current == peoples_scores.end())
{
  peoples_scores.insert({name,
    best_scores.insert({score, name})});
}
else if (current->second->first < score)
{
  best_scores.erase(current->second);
  current->second =
    best_scores.insert({score, name});
}
```

I don't think this makes any functional difference to the implementation.

In `print_scores()`, the `for` loop is not idiomatic, and this is where I think the problem may be. `best_scores` needs to be iterated in reverse order. The correct way to do this is to use the reverse iterators from begin to end, and in fact the `const` versions of these can be used:

```
const auto end_it = best_scores.crend();
for (auto it = best_scores.crbegin();
   it != end_it;
   ++it)
{
  std::cout << it->second << ": "
    << it->first << '\n';
}
```

There is no range based `for` loop for reverse iterators in the C++ language itself yet. With a bit of refactoring, `std::for_each` could be used instead:

```
#include <algorithm>
struct output_score
{
  void operator()(
    const std::pair<int, std::string>& score)
  {
    std::cout << score.second << ": "
      << score.first << '\n';
  }
};
void print_scores()
{
  std::for_each(best_scores.crbegin(),
          best_scores.crend(),
          output_score());
}
```

The original implementation uses forward iterators from end to begin, with a post decrement in the condition. As observed, this does in fact work, but there is a potential problem with the terminating step of the loop. This decrements the iterator to a position before `best_scores.begin()`, which does not exist, and technically this is undefined behaviour. Because the iterator is never dereferenced when it reaches this invalid position, I expect you would usually get away with this implementation, but maybe this is the cause of "occasionally odd things happen ...". I haven't been able to find anything else.

James Holland < james.holland@babcockinternational.com>

It may be the case that the student's code runs without error on a particular system. This would be unfortunate as there are two features of the program that rely on undefined behaviour. The program may well fail on other platforms. The first feature that results in undefined behaviour is located in `add_score()` and is the comparison of default-constructed iterators. One iterator is created by `operator[]()` belonging to `peoples_scores`. The other is the global `none`.

The fact that the standard does not permit the comparison of default constructed iterators is a pity as it renders invalid what would otherwise be a perfectly good function. Perhaps later versions of the standard will permit such comparisons. I could not think of a reasonable way of modifying the student's code while still employing `operator[]()`. Instead, I adopted a fairly straightforward approach that first searches `peoples_scores` for a record with a key of name and then adds a record or modifies the existing record accordingly.

The second undefined feature is located in `print_scores()` and results in an iterator pointing to one element before the start of `best_scores`. This is also not permitted under the standard. Probably the simplest way of resolving this problem is to rewrite the loop using a range-based `for` loop. The original loop was designed to print `best_scores` starting with the last record. The range-based `for` loop can only iterate through the container starting from the first record. The desired result can be obtained by storing the elements of `best_scores` with the greatest key value at the start of the container. This is achieved by changing the third template parameter of the `best_scores` type to `std::greater<int>`. The best scores will now be printed starting with the highest value.

One slight deficiency with the program that still exists is that people's names are stored twice; once in `peoples_scores` and once in `best_scores`. It would be more efficient for `best_scores` to refer the names already stored in `peoples_scores`. This is a little fiddly to achieve and I do not attempt the modification here.

Finally, I present my version of the student's code below.

```
#include <iostream>
#include <map>
#include <sstream>
#include <unordered_map>
using best_scores_type = std::multimap<
  const int, const std::string,
   std::greater<int>>;
best_scores_type best_scores;
std::unordered_map<std::string,
  best_scores_type::const_iterator>
  peoples_scores;
void add_score(const std::string & name,
  const int score)
{
  const auto people_position =
    peoples_scores.find(name);
  if (people_position != peoples_scores.end())
  {
    if (score > people_position->
                  second->first)
    {
      best_scores.erase(
        people_position->second);
      people_position->second =
        best_scores.insert({score, name});
    }
  }
  else
  {
    const auto best_scores_position =
      best_scores.insert({score, name});
    peoples_scores.insert({name,
      best_scores_position});
  }
}
void print_scores()
{
  for (const auto & person : best_scores)
  {
    std::cout << person.second << ": "
      << person.first << '\n';
  }
}
int main()
{
```

```
  for (;;)
  {
    std::cout << "Enter name and score: ";
    std::string lbufr;
    if (!std::getline(std::cin, lbufr)) break;
    std::string name;
    int score;
    std::istringstream(lbufr) >> name
      >> score;
    add_score(name, score);
  }
  std::cout << "\nBest scores\n";
  print_scores();
}
```

Herman Pijl <herman.pijl@telenet.be>

Apparently the developer wants to keep track of the high scores and simultaneously maintain a lookup that maps the user to an entry in the high score table.

The **print_scores** function looks a bit suspicious.

I decide to run the program, and after the prompt appears, I press Ctrl-d to end the input. The result is a segmentation fault (I am running on Cygwin). Iterating over the elements of a container in reverse order should not be an adventure. Just use the reverse iterators that were designed exactly for this purpose (**rbegin()** and **rend()**) to avoid problems or even crashes as the one I had.

Reverse iteration doesn't look natural at first sight. The ordered containers allow to specify an ordering criterion. The default ordering criterion happens to be **std::less**, but nothing keeps you from using another ordering criterion from the algorithm library, e.g. **std::greater**. The given code explicitly mentions the default ordering criterion and this explicit (default) choice is probably meant to be a hint to change the program. Ordering from high to low values allows to iterate over the container with the usual **begin()** and **end()** boundaries.

Let's move to the next problem in this code. The **peoples_scores** map attempts to keep track of the position in the **best_scores** map. Unfortunately it is not such a good idea to keep a copy of an iterator in a table. Stroustrup (4th Ed. $31.2) mentions that the ordered [multi]map/sets are usually implemented as balanced binary trees (mostly red-black trees). To keep such a tree balanced (and therefore performant O(log(n))) some pruning is needed causing the traversal (=iteration) to be different after insertions/deletions.

So if you cannot use an iterator, then what can you use? Remarkably the answer is: a pointer! There is no such thing as a commandment that says: "Thou shalt not use pointers!" Some people try to avoid them dogmatically, but sometimes you have no choice.

The standard library does this too in some cases, e.g. when you construct an **istream_iterator**, you typically pass an **istream** to the constructor as reference, BUT the **istream** iterator takes the address of that **istream** (thus a pointer) and it keeps that as a private member. One of the good reasons to keep a pointer is that the **istream_iterator** is an input iterator ($33.1.2) and as such it has to be copyable.

I am going to keep the **const_pointer** type of the multimap, i.e. **std::multimap<int, std::string>::const_pointer** will become my entry type.

The **value_type** of a map is typically a (K, V) pair.

I assume that the standard allocator will allocate the nodes on the heap, so that the addresses of the **value_type** will not change when a red-black tree gets reorganised.

The **best_scores** and **peoples_scores** maps have to be maintained 'transactionally', so that there is always exactly one entry in both for a particular user.

In order to add an entry to the **best_scores** map, I use the **emplace** method. As it returns an iterator, I deference it and use the address-of operator to find my **const_pointer**:

```
&*best_scores.emplace(score, name);
```

When there is no entry for the user I can just do:

```
peoples_scores.emplace(name,
  &*best_scores.emplace(score, name));
```

When there is already an entry for the user, then I have to check whether the score is better than the previous score. If that is the case, then I have to erase the previous score. Unfortunately, multiple users can have the same score. So in order to find the right entry and erase it I use

```
// bs is reference to best_scores
bs.erase(std::find_if(
  bs.lower_bound(prevScore),
  bs.upper_bound(prevScore),
  [&](auto & it)-> bool{
    return it.second == name;
}));
```

After that I need to add a new entry in the **best_scores** table and update the reference to it in the **peoples_scores** map.

```
itFind->second = &*bs.emplace(score, name);
```

It seems to work fine.

Further enhancements could be the introduction of a good old fashioned struct containing name and score and defining the extraction operator **>>** on it so that so that you can write the processing as a **for_each** call.

A complete solution is shown below:

```
#include <functional>
#include <iostream>
#include <map>
#include <sstream>
#include <unordered_map>
// added
#include <algorithm>
#include <cassert>
#include <iterator>
// Best scores
/*added*/ std::multimap<int, std::string,
  std::less<>> typedef best_scores_type;
std::multimap<int, std::string,
  std::less<>> best_scores;
auto & bs = best_scores;
// new unordered map
std::unordered_map<std::string,
  best_scores_type::const_pointer>
  nps;//new_peoples_scores
void new_add_score(std::string name,
  int score)
{
  auto itFind = nps.find(name);
  if (itFind != nps.end()){
    // player already present
    int prevScore = itFind->second->first;
    if (prevScore < score){
      bs.erase(std::find_if(
        bs.lower_bound(prevScore),
        bs.upper_bound(prevScore),
        [&](auto & it)-> bool{
          return it.second == name;}));
      itFind->second =
        &*bs.emplace(score, name);
    }
  } else { // new player
    nps.emplace(name,
      &*bs.emplace(score, name));
  }
}
void print_scores()
{
  std::for_each(best_scores.rbegin(),
    best_scores.rend(),
```

```
    [](auto & it){
      std::cout << it.second << ": "
        << it.first << '\n';});
}
struct NameScoreEntry{ std::string name;
  int score; };
std::istream & operator>>(std::istream& is,
  NameScoreEntry& entry)
{
  is >> entry.name >> entry.score;
  return is;
}
void extract(std::istream & is){
  std::istream_iterator<NameScoreEntry>
    ist(is), eos;
  std::for_each(ist, eos, [&](auto i){
    new_add_score(i.name, i.score);});
}
int main(){
  std::cout <<
    "Enter name and score (multiple lines): ";
  extract(std::cin);
  print_scores();
}
```

Commentary

This is a critique that is principally about undefined behaviour – often abbreviated to UB. There are in fact *three* different examples of this in the code presented. You might want to stop now and see if you can find them with this hint before reading on!

The first problem is the use of the default-constructed **`entry`** object named **`none`**. As James noted, the standard does not allow comparisons between default-constructed iterators and so the comparison

```
if (current != none)
```

is not valid. As it happens, the code appears to run successfully with a number of combinations of compilers and flags: one of the problems with undefined behaviour is that it can be quite hard to detect.

The second problem is that the loop in **`print_scores`** counting backwards through **`best_scores`** decrements the iterator **`it`** to point *before* the beginning of the collection. This is also not valid, and may cause runtime failures depending on exactly how **`multimap`** is implemented in the standard library being used. Some years ago I encountered a similar problem with code that removed items from a **`std::list`** and decremented the iterator referring to the item being removed. This had worked for years with one implementation of the standard library, but when ported to a different environment decrementing the iterator when the item being deleted was at **`begin()`** caused failures at runtime.

The third problem is that the **`typedef`** for **`entry`** is incorrect. It refers to an iterator into a **`multimap`** with a defaulted comparator but the actual map has an explicitly specified comparator of **`std::less<>`**. This subtle difference means the resultant iterators are of different types and adds some more possibly troubling behaviour.

(Note: **`std::less<>`** was added in C++ 14, which is why Chris Main couldn't compile the code using -std=c++11. It was proposed by Stephan Lavavej and his proposed wording was voted into the standard without needing *any* editing, which is very unusual!)

There are several ways to attack these problems; but *identifying* them is often the hardest step. Fortunately, many compilers provide extra validation when using the standard library that can make this easy.

For example, compiling the sample code using gcc and adding the flag **`-D_GLIBCXX_DEBUG`** will use an instrumented version of the standard library. The code as presented won't even compile with this flag specified because of the mismatched iterator types.

Similar instrumentation is available with MSVC when using the debugging runtime library (eg adding **`-MDd`**).

Incidentally, this is another reason to prefer using the standard library to your own hand-written code…

As the entries showed, solving the problem caused by trying to use none as a sentinel value can be resolved by the use of **`find`** rather than the simple use of the square bracket operator.

The over-enthusiastic decrementing of the iterator can be resolved by using a more idiomatic loop: either by changing the comparator to **`std::greater<>`** or by changing the loop to use **`crbegin`** and **`crend`**. The first option has the additional benefit of allowing use of range-for.

I was expecting someone to comment about the placement of **`typedef`** in the middle of the line: placing **`typedef`** at the beginning of the line is an extremely common coding convention. In modern C++ code though I would recommend **`using`** over **`typedef`** as I think it is more readable.

The winner of CC 103

I was (deliberately) a little vague in setting the critique about what exactly the symptoms were ("occasionally odd things happen") – perhaps I should have been a little more forthcoming! Unfortunately, of course, when undefined behaviour occurs the symptoms are often just like this…

While Chris did not identify the undefined behaviour, his instincts for writing idiomatic code were sound and so his modifications did in fact deal with the two main pieces of UB.

James identified the two main sources of UB and he also, by changing the comparator, was able to change the code in **`print_scores`** to use range-for, which makes it clearer.

Herman was very close to identifying the problem; but in fact the iterators are not invalidated when the map is re-balanced. His replacement of iterators with pointers was valid and did have the side-effect of removing the undefined behaviour. I did like his introduction of a simple helper struct **`NameScoreEntry`** to assist with reading the data in: it is very easy to create such structs in C++ and there can be significant benefits for readability with having named fields.

Overall, I think James provided the best critique, so I have awarded him the prize.

Listing 2

```
#include <iterator>
#include <vector>
// get unique values in the range [one, two)
template <typename iterator>
std::vector<typename iterator::value_type>
unique(iterator one, iterator two)
{
  if (distance(one, two) < 2)
  {
    // no duplicates
    return {one, two};
  }
  // first one can't be a duplicate
  std::vector<typename iterator::value_type>
  result{1, *one};
  while (++one != two)
  {
    auto next = *one;
    bool is_unique =
     (*result.rbegin() != next);
    if (is_unique)
    {
       result.push_back(next);
    }
  }
  return result;
}
```

Troy Hunt: An Interview

Emyr Williams continues the series of interviews with people from the world of programming.

Troy Hunt is based in Gold Coast, and is a Microsoft Regional Director as well as an MVP (Microsoft Valued Professional) in the field of developer security, and has become a world class security consultant. He has travelled the globe giving training lectures on security for software engineers. As well as being a Pluralsight author, he is also the man behind the website haveibeenpwned.com where a user can enter their e-mail and check if they have an account on a site that's been compromised. His blog can be found at www.troyhunt.com

How did you get in to computer programming? Was it a sudden interest? Or was it a slow process?

> I had a curiosity as a young teenager but frankly, I preferred to be outdoors doing something active. It wasn't until I turned 14 and we moved from Australia to the Netherlands (which is often not very conducive to outdoor activities!) that I began showing more interest in computing.

What was the first program you ever wrote? And in what language was it written in?

> It would have been something in BASIC but I honestly can't remember what. Most of my code exposure then was hacking around games and other subversive activities.

What would you say is the best piece of advice you've been given as a programmer?

> I can't think of one piece of advice specifically, but a friend I worked with many years ago pushed me into blogging and that then opened up many opportunities that leveraged my coding experience and led to where I am today.

How did you get in to the field of software security? Was this part of your day job when you worked at Pfizer? Or did it occur naturally?

> I worked as a software architect at Pfizer responsible for how we delivered solutions across Asia Pacific and whilst security was also a component of that, it wasn't the sole focus of the role. But what the role did is exposed me to many seriously bad security practices; Pfizer outsourced everything to vendors in low cost markets and had a very strong focus on price so you can imagine some of the security transgressions that led to!

EMYR WILLIAMS

Emyr Williams is a C++ developer who is on a mission to become a better programmer. His blog can be found at www.becomingbetter.co.uk

Code Critique Competition 104 (continued)

Code critique 104

(Submissions to scc@accu.org by April 1st)

I was trying to write a simple template that gets the unique values from a range of values (rather like the Unix program uniq) but my simple test program throws up a problem.

```
test with strings
a a b b c c => a b c
test with ints
1 1 2 2 3 3 => 1 1 2 3
```

Why is the duplicate 1 not being removed?

The code is in Listing 2 (unique.h) and Listing 3 (unique.cpp).

You can also get the current problem from the accu-general mail list (next entry is posted around the last issue's deadline) or from the ACCU website (http://accu.org/index.php/journal). This particularly helps overseas members who typically get the magazine much later than members in the UK and Europe.

Listing 3

```
#include <iostream>
#include <string>
#include <vector>
#include "unique.h"
template <typename T>
void test(std::ostream &os,
  std::vector<T> const &vector)
{
   auto result =
     unique(vector.begin(), vector.end());
   auto out =
     std::ostream_iterator<T>(os, " ");
   copy(vector.begin(), vector.end(), out);
   os << "=> ";
   copy(result.begin(), result.end(), out);
   os << "\n";
}
int main()
{
   std::cout << "test with strings\n";
   std::vector<std::string> ptrs;
   ptrs.push_back("a");
   ptrs.push_back("a");
   ptrs.push_back("b");
   ptrs.push_back("b");
   ptrs.push_back("c");
   ptrs.push_back("c");
   test(std::cout, ptrs);

   std::cout << "test with ints\n";
   std::vector<int> ints;
   ints.push_back(1);
   ints.push_back(1);
   ints.push_back(2);
   ints.push_back(2);
   ints.push_back(3);
   ints.push_back(3);
   test(std::cout, ints);
}
```

Listing 3 (cont'd)

ACCU Information

Membership news and committee reports

View from the Chair

Bob Schmidt
chair@accu.org

In my last column I mentioned that I was scheduled to participate in the **Hour of Code** program. Hour of Code is a "global movement reaching tens of millions of students in 180+ countries" [1] sponsored by **code.org**, a "non-profit dedicated to expanding access to computer science, and increasing participation by women and under-represented minorities." [2]

I volunteered for Hour of Code in 2015, but was not contacted to participate that year. Last year, Dr. Kathleen Neuber of St. Charles School here in Albuquerque asked if I would come to her school and talk to her students.

I spent two hours the morning of December 5th at the school, interacting with approximately 50 students ranging in age from ten to 13 years. As I worked my way through the slides I had prepared, I talked about how I had never seen a computer until I was 18 years old and in my second college semester, and how that compares to today's students who grow up with a computer in their hands.

I explained that the type of systems on which I work, from large-scale process control systems to very-small-scale embedded systems, are somewhat different from the type of applications they are used to using on their phones and tablets. I gave a brief introduction to closed-loop control, using a heating thermostat as a simple example.

I spent a little time talking directly to the young ladies in the audience. I explained that women are under-represented in our field, and that there are several reasons why – including peer pressure and bullying. I encouraged the young ladies, if they are interested in a career in one of computer industries, to push back against those pressures. I explained that diverse project teams have been shown to be more successful than those that aren't. I showed a picture of a computer system I helped develop, install, and

Troy Hunt: An Interview (continued)

I see from your blog that some folks send you details of breaches, such as data dumps etc? But do you carry out tests yourself on websites or web services? And if so, how do you do that within the bounds of staying legal? Do you poke and prod at online systems looking for holes and attack vectors?

> I make a real point of doing data breach verification in a very transparent way; I expect that people may be watching and I'm exceptionally cautious to ensure I remain ethical the entire time. I look for publicly observable patterns of vulnerable coding, things like missing HTTPS, email address enumeration risks, improperly configured servers, risky patterns with cookies etc. I try and answer the question "does this look like the sort of site that would be vulnerable to attack?". I also reach out to HIBP subscribers in the alleged data breach and simply ask them – "Did you use that service and is this your data". That's a very reliable means of verification.

Having read your blog for some time now, I get the impression you're insanely busy. How do you maintain the balance between work and family life?

> We all trade off work and families. Many people don't like to think about it that way, but we put a price on our families every day we go to work even in very traditional jobs. The balance my wife and I strike is that I travel a lot and work long hours at home too, but I have heaps of flexibility. I'm often taking the kids to school, I always watch them at tennis, my wife and I often go out for lunch and of course we enjoy the rewards of a successful career too. It wouldn't work for many other people, but it's a balance that works well for us.

What would be an average work day for you?

> Who knows! If I'm at home, getting up by 5am and sitting outside near the water with a coffee doing emails and catching up on everything that's happened overnight for an hour or 2. After that I'm often having meetings about upcoming events or courses, writing new blog material, working on HIBP and doing any number of other things. When I'm travelling, it's non-stop going between hotels, airports, conferences and workshops. That's a really intense period that often takes me weeks to unwind from once I'm home.

If you were to start your career again now, what would you do differently? Or if you could go back to when you started programming what would you say to yourself?

> It's a different world now to when I was starting out. My first year at uni was the first time I saw the internet so I never really had the advantage of all the online resources we have today. If I'd had them, I would have done a heap of learning online through resources like Pluralsight and built up an online identity much earlier. That's been the most valuable thing for me in later years and what I wish I had have done earlier.

What would you say is one of the most important books you've read as a developer?

> *HTML for dummies*. Seriously, I still have the book I bought in '95 and used that to start building web apps. It's such a trivial thing by today's standards, but that's what helped get me started in a time where information was sparse.

What would you describe as the biggest "ah ha" moments, or surprises you've come across when you're chasing down a bug?

> I'm continually surprised at how on earth some code ever worked to begin with! We all do this – look back at the things we wrote with wonderment – and I do the same thing on a regular basis.

Do you have any regrets? Such as followed a different technical route or something like that?

> I'm really happy with my life at present so it's hard to have regrets as all the things I've done have led me to where I am now. Perhaps the biggest would be not starting down the online identity path earlier. I wish I'd been able to fast track the path to my current life but even then, other environmental factors may not have made that happen any earlier. I spend very little time regretting things, I'm usually looking forward to the next thing.

Do you mentor other developers? Or did you ever have a mentor when you started programming?

> 12. I'm not overly fond of the premise of mentoring as some sort of discrete process. I prefer the idea of having many people you talk to, draw inspiration from and give advice to. Of course I've had those relationships more with some people than others, my suggestion to people is to surround yourself (either physically or virtually) with people you respect and aspire to be like and that's certainly had a very positive impact on me.

Finally, what advice would you give to someone is looking to start a career as a programmer?

> Build stuff. More than anything, experience counts *massively* towards your future potential. Go and learn from resources like Pluralsight then start a project or contribute to open source projects or do *something* that actually produces an end product. As someone who's interviewed a lot of people before, I care very little about what school they went to or the grades they got and I care *massively* about what they're actually able to do.

maintain starting in 1981, and which ran until it was replaced in 2013. In the foreground of the picture was a young lady who was at the time younger than the computer system. I explained that she was the software project lead on that system's replacement project, and was supervising my work as part of my current contract.

I also brought samples of some of the embedded system products I've produced over the past 20 years, including prototypes and production runs of the printed circuit boards (PCBs), and the schematics that were used to generate the PCB layouts. I had some spare boards that the kids could pass around so they could get an up-close look.

We talked about areas of computing that I've seen in the literature – including Big Data – that are predicted to be high growth areas (and used pictures of the Large Hadron Collider as an example, generating a petabyte of data per year when it is running).

The older students are participating in the Future City competition [3], so I talked a bit about where computer systems might be applied in cities the future. I also opined that we aren't very good at predicting the future, and used as an example that when I wrote my first program in 1978 (in FORTRAN, for an IBM 360), nobody (that I knew of, at least) was predicting we'd all be carrying around computers in our pockets.

I was worried I was going to blow through the slides, but I ended up talking for longer than I expected. When it came time for questions, most of the students were quiet, but there was one young lady who asked multiple questions. She stayed behind after the rest of the students had gone back to their regularly scheduled classes, and we talked one-on-one for a short time. I gave her one of the sample boards that had been passed around.

Why spend so much ink on this? Not long ago, Russel Winder (our conference chair) asked what we (ACCU) could do to support computing in schools, and we discussed the issue in our January committee meeting. The answer to the question is, as an *organization*, 'not much', because ACCU as an *organization* does not have a lot of unallocated resources; however, as a collection of professionals, we are rich in individual resources, and we each have the potential to contribute.

One thing ACCU has done is to adopt **Code Club** [4] as one of our charities, and we anticipate they will have a presence at the conference this year. Several of our members are active with the group. Russel mentioned **Computing at School**, another organization "promoting and supporting excellence in computer science education".[5] Hour of Code is another option. I encourage you to consider participating in one of these fine programs, and if you do, consider writing up your experience and share it with the rest of us. If I can do it, so can you.

Member news

We are starting a new section in *CVu* this issue – Member News. This is an opportunity for our members to let us know what's happening in their professional lives. This is a work-in-progress, but the general rules are as follows:

- The section is available to members only;
- News should be of a professional nature: new jobs, promotions, completion of major projects, release of a product from a company owned by the member, etc.;
- The news should be member-oriented, not company-oriented;
- The editor of *CVu* has the final say over what is published, and may edit submissions for length and content.

ACCU Conference 2017

The 2017 ACCU conference is scheduled for Wednesday, April 26th through Saturday, April 29th, with pre-conference tutorials on Tuesday the 25th [6]. The schedule for the conference has been announced, and is available on our website [7].

Election of officers

The ACCU Annual General Meeting will be held in conjunction with the conference, on Saturday April 29th. These are the important dates associated with the election:

Announcement	29 January, 2017 (90 days before AGM)
Proposal deadline	28 February, 2017 (60 days before AGM)
Draft agenda	18 March, 2017 (42 days before AGM)
Agenda freeze	1 April, 2017 (28 days before AGM)
Voting opens	8 April, 2017 (21 days before AGM)

We will once again be utilizing on-line voting for the election. Details will be posted to accu-members and accu-announce.

Independent-of-the-committee spotlight

Those of you who have been keeping score know that, for the past 8 months, I have been asking for a volunteer to take my place as Auditor for the second year of my term. I'm pleased to announce that Niall Douglas has volunteered for the role. Niall has impeccable qualifications for the role, with an education and business experience in accounting. Please join me in thanking Niall for volunteering.

Call for volunteers

We finally have an auditor, but we still have several opportunities available for volunteering:

- The ACCU web site uses Xaraya, a PHP framework that has been moribund for the last 4 years at least, and a replacement is overdue.
- Martin Moene has informed the committee that he will be stepping down as our web site editor effective July 1st. Please join the committee and me in expressing our thanks for all of his contributions over his years of service. (I will have more to say on this in the next issue.) This means we are not only still looking for someone to assist with web site admin duties; we now are also looking for a volunteer to take over the web site editor duties.

Please contact me if you are interested in any of these positions.

I'd like to thank Dr. Neuber and her students at St. Charles School for inviting me to participate in their Hour of Code. I had a great time interacting with the students.

References

[1] Hour of Code: https://hourofcode.com/us
[2] code.org: https://code.org/about
[3] Future City Competition: http://futurecity.org/
[4] Code Club: https://www.codeclub.org.uk/
[5] Computing at School: https://www.computingatschool.org.uk
[6] ACCU 2017: https://conference.accu.org/site/index.html
[7] ACCU 2017 Schedule: https://conference.accu.org/site/stories/2017/schedule.html

Member news

Pavol Rovensky (Devon, UK) has released version 1.0 of ProudNumbers, through the company he founded, Hexner Limited.

ProudNumbers is a management account report generator for Sage 50 accounting software. The program is designed to generate Management Account reports for various time periods and enables full customisation of the chart of accounts for reporting purposes. The reports are generated in seconds. The generated output is in spreadsheet or PDF format and facilitates communication between accountants and their clients.

Pavol spent more than 5 years developing ProudNumbers. He has been a member of ACCU since 2007. (www.hexner.co.uk)

"The magazines"

The ACCU's *C Vu* and *Overload* magazines are published every two months, and contain relevant, high quality articles written by programmers for programmers.

"The conferences"

Our respected annual developers' conference is an excellent way to learn from the industry experts, and a great opportunity to meet other programmers who care about writing good code.

"The community"

The ACCU is a unique organisation, run by members for members. There are *many* ways to get involved. Active forums flow with programmer discussion. Mentored developers projects provide a place for you to learn new skills from other programmers.

"The online forums"

Our online forums provide an excellent place for discussion, to ask questions, and to meet like minded programmers. There are job posting forums, and special interest groups.

Members also have online access to the back issue library of ACCU magazines, through the ACCU web site.

ACCU | JOIN: IN

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

Invest in your skills. Improve your code. Share your knowledge.

Join a community of people who care about code. Join the ACCU.

Use our online registration form at **www.accu.org**.

Design: Pete Goodliffe

QBS Software Ltd is an award-winning software reseller and Intel Elite Partner.

To find out more about Intel products please contact us:

020 8733 7101 | **enquiries@qbssoftware.com**
www.qbssoftware.com/parallelstudio

qbs SOFTWARE