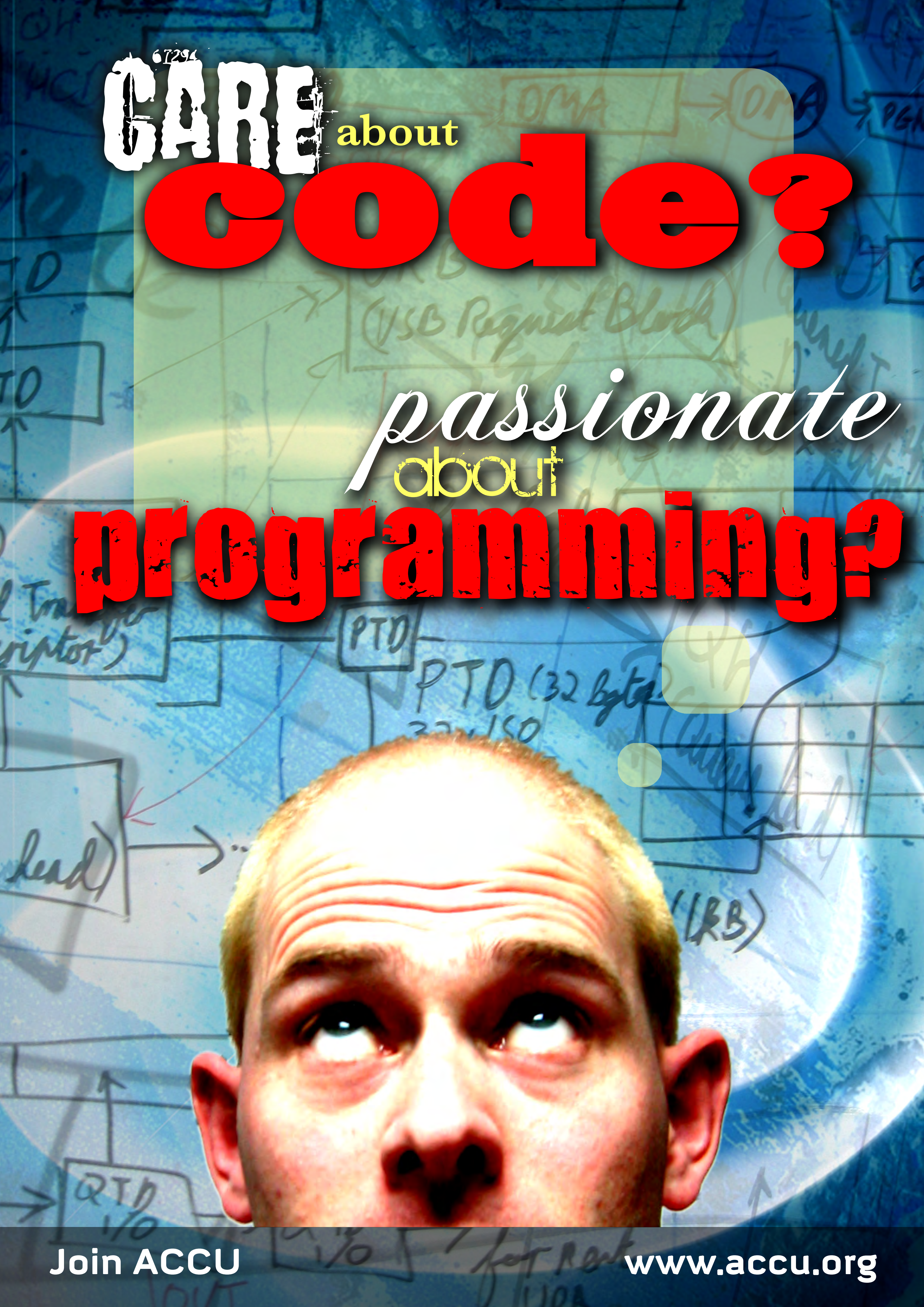
the magazine of the accu

www.accu.org

{cvu}

Volume 29 • Issue 4 • September 2017 • £3


CARE about
code?
passionate
about
programming?
Join ACCU
www.accu.org

{cvu}

Volume 29 Issue 4
September 2017
ISSN 1354-3164
www.accu.org

Editor
Steve Love
cvu@accu.org

Contributors
Silas S. Brown, Pete Cordell, Francis Glassborow, Pete Goodliffe, Baron M, Roger Orr

ACCU Chair
chair@accu.org

ACCU Secretary
Malcolm Noyes
secretary@accu.org

ACCU Membership
Matthew Jones
accumembership@accu.org

ACCU Treasurer
R G Pauer
treasurer@accu.org

Advertising
Seb Rose
ads@accu.org

Cover Art
Pete Goodliffe

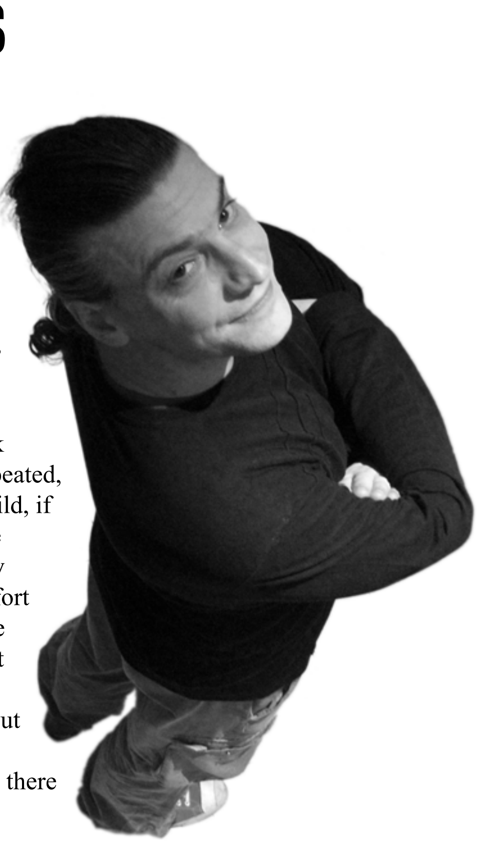
Print and Distribution
Parchment (Oxford) Ltd

Design
Pete Goodliffe

accu

The art of laziness

I'm often moved to remark that being lazy, as a programmer, can be a virtue. I see it as a (rather trite, I admit) variation on the equally clichéd adage that it's better to work smarter than to work harder. The goal of laziness is to avoid the need to work harder at some point in the future, rather than to **evade** working now. In that respect, the distinction is very much like the difference between tax avoidance and tax evasion, although with less drastic consequences for contravening the laws that apply.



A simple example of avoiding (unnecessary) work would be to automate something that has to be repeated, such as a software deployment script or even a build, if they are tasks with several steps. It's easy to make mistakes when doing these things manually, partly because they're generally boring to do. Putting effort in to creating a script that can be run with a simple command-line (or a single click) saves time, effort and possibly embarrassment later. For complex tasks, the effort of automating can be significant, but the payoff is commensurately large, because the more complicated a thing is, the more opportunity there is to get it wrong.

A simple example of evading work would be to copy some code (say from the Internet) that does almost what you need right now, and then to bang it with a hammer until it's just right. A more subtle version would be to use some freely available third-party library, without paying sufficient attention to how active the code-base is, and without considering the burden of how and when upgrades should be introduced, and the testing that goes with that. You might have to fix any bugs you discover yourself, and even if you do not need to re-publish the fixes you make, there is the added burden of merging changes back to your own version of the code.

Doing something because it's expedient might not necessarily be beneficial in the long run. There is a fine line between convenience and imprudence. If you want to be lazy, it's worth making the effort to do it right.

Steve

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

SUBMISSION DATES

C Vu 29.5:	1st October 2017
C Vu 29.6:	1st December 2017
Overload 141:	1st November 2017
Overload 141:	1st January 2018

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

Navigating a Route

Pete Goodliffe helps us work with unfamiliar code.

...the Investigation of difficult Things by the Method of Analysis ought ever to precede the Method of Composition.
~ Sir Isaac Newton

A new recruit joined my development team. Our project, whilst not vast, was relatively large and contained a number of different areas. There was a lot to learn before he could become effective. How could he plot a route into the code? From a standing start, how could he rapidly become productive?

It's a common situation; one which we all face from time to time. If you don't, then you need to see more code and move on to new projects more often. (It's important not to get stale from working on one codebase with one team forever.)

Coming into any large existing codebase is hard. You have to rapidly:

- Discover where to start looking at the code
- Work out what each section of the code does, and how it achieves it
- Gauge the quality of the code
- Work out how to navigate around the system
- Understand the coding idioms, so your changes will fit in sympathetically
- Find the likely location of any functionality (and the consequent bugs caused by it)
- Understand the relationship of the code to its important satellite parts (e.g., its tests and documentation)

You need to learn this quickly, as you don't want your first changes to be too embarrassing, accidentally duplicate existing work, or break something elsewhere.

A little help from my friends

My new colleague had a wonderful head start in this learning process. He joined an office with people who already knew the code, who could answer innumerable small questions about it, and point out where existing functionality could be found. This kind of help is simply invaluable.

If you are able to work alongside someone already versed in the code, then exploit this. Don't be afraid to ask questions. If you can, take opportunities to pair program and/or to get your changes reviewed.

Your best route into code is to be led by someone who already knows the terrain. Don't be afraid to ask for help!

If you can't pester people nearby, don't fear; there may still be helpful people further afield. Look for online forums or mailing lists that contain helpful information and helpful people. There is often a healthy community that grows around popular open source projects.

The trick when asking for help is to always be polite, and to be grateful. Ask sensible, appropriate questions. "Can you do my homework for me?" is never going to get a good response. Always be prepared to help others out with information in return.

Employ common sense: make sure that you've Googled for an answer to your question first. It's simple politeness to not ask foolish questions that you could easily research yourself. You won't endear yourself to anyone if you continually ask basic questions and waste people's precious time. Like the boy who cried wolf and failed to get help when he really needed it, a series of mind-numbingly dumb questions will make you less likely to receive more complex help when you need it.

Look for clues

If you are rooting in the murky depths of a software system without a personal guide, then you need to look for the clues that will orient you around the code.

These are good indicators:

Ease of getting the source

How easy is it to obtain the source?

Is it a single, simple checkout from version control that can be placed in any directory on your development machine? Or must you check out multiple separate parts, and install them in specific locations on your computer?

Hardcoded file paths are evil. They prohibit you from easily building different versions of the code.

Healthy projects require a single checkout to obtain the whole codebase, and the code can be placed in *any* directory on your build machine. Do not rely on multiple checkout steps, or code in hardcoded locations.

As well as availability of the source code itself, consider availability of *information about* the code's health. Is there a CI (*continuous integration*) build server that continually ensures that all parts of the code build successfully? Are there published results of any automated tests?

Ease of building the code

This can be very telling. If it's hard to build the code, it's often hard to work with it.

Does the build depend on unusual tools that you'll have to install? (How up-to-date are those tools?)

How easy is it to build the code from scratch? Is there adequate and simple documentation in the code itself? Does the code build straight out of source control, or do you first have to manually perform many small configuration tweaks before it will build?

Does one simple, single step build the entire system, or does it require many individual build steps? Does the build process require manual intervention? [1] Can you work on a small part of the code, and only build that section, or must you rebuild the whole project repeatedly to work on a small component?

A healthy build runs in one step, with no manual intervention during the build process.

How is a release build made? Is it the same process as the development builds, or do you have to follow a very different set of steps?

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

When the build runs, is it quiet? Or are there many, many warnings that may obscure more insidious problems?

Tests

Look for tests: unit tests, integration tests, end-to-end tests, and the like. Are there any? How much of the codebase is under test? Do the tests run automatically, or do they require an additional build step? How often are the tests run? How much coverage do they provide? Do they appear appropriate and well constructed, or are there just a few simple stubs to make it look like the code has test coverage?

There is an almost universal link here: code with a good suite of tests is usually also well factored, well thought out, and well designed. These tests act as a great route into the code under test, helping you understand the code's interface and usage patterns. It's also a great place from which to start working on a bugfix (you can start by adding a simple, failing unit test—then fix that test, without breaking the others).

File structure

Look at the directory structure. Does it match the code shape? Does it clearly reveal the areas, subsystems, or layers of the code? Is it neat? Are third-party libraries neatly separated from the project code, or is it all messily intermingled?

Documentation

Look for the project documentation. Is there any? Is it well written? Is it up-to-date? Perhaps the documentation is written in the code itself using *NDoc*, *Javadoc*, *Doxygen*, or a similar system. How comprehensive and up-to-date does this documentation appear?

Static analysis

Run tools over the code to determine the health and to plot out the associations. There are some great source navigation tools available, and Doxygen can also produce very usable class diagrams and control flow diagrams.

Requirements

Are there any original project requirements documents or functional specifications? (In my experience, these often tend to bear little relation to the final product, but they are interesting historical documents nonetheless.) Is there a project wiki where common concepts are collected?

Project dependencies

Does the code use specific frameworks and third-party libraries? How much information do you need to know about them? You can't learn every aspect of all of them initially, especially because some libraries are huge (Boost, I'm looking at you). But it pays to get a feel for what facilities are provided for you, and where you can look for them.

Does the code make good use of the language's standard library? Or do many wheels get reinvented? Be wary of code with its own set of custom collection classes or homegrown thread primitives. System-supplied core code is more likely to be robust, well tested, and bug-free.

Code quality

Browse through the code to get a feel for the quality. Observe the amount and the quality of code comments. Is there much dead code—redundant code commented out but left to rot? Is the coding style consistent throughout?

It's hard to draw a conclusive opinion from a brief investigation like this, but you can quickly get a reasonable feel for a codebase from some basic reading.

Architecture

By now you should be able to get a reasonable feel for the shape and the modularisation of the system. Can you identify the main layers? Are the layers cleanly separated, or are they all rather interwoven? Is there a database layer? How sensible does it look? Can you see the schema? Is it sane? How does the app talk to the outside world? What is the GUI technology? The file I/O tech? The networking tech?

Ideally, the architecture of a system is a top-level concept that you learn before digging in too deeply. However, this is often not the case, and you *discover* the real architecture as you delve into the code.

Often the *real* architecture of a system differs from the *ideal* design. Always trust the code, not the documentation.

Perform *software archaeology* on any code that looks questionable. Drill back through version control logs and 'svn blame' (or the equivalent) to see the origin and evolution of some of the messes. Try to get a feel for the number of people who worked on the code in the past. How many of them are still on the team?

Learn by doing

A woman needs a man like a fish needs a bicycle.
~ Irina Dunn

You can read as many books as you like about the theory of riding a bicycle. You can study bicycles, take them apart, reassemble them, investigate the physics and engineering behind them. But you may as well be learning to ride a fish. Until you get on a bicycle, put your feet on the pedals and try to ride it for real, you'll never advance. You'll learn more by falling off a few times than from days of reading about how to balance.

It's the same with code.

Reading code will only get you so far. You can only really learn a codebase by getting on it, by trying to ride it, by making mistakes and falling off. Don't let inactivity prevent you from moving on. Don't erect an intellectual barrier to prevent you from working on the code.

I've seen plenty of great programmers initially paralysed through their own lack of confidence in their understanding.

Stuff that. Jump in. Boldly. Modify the code.

The best way to learn code is to modify it. Then learn from your mistakes.

So what should you modify?

As you are learning the code, look for places where you can immediately make a benefit, but that will minimise the chances you'll break something (or write embarrassing code).

Aim for anything that will take you around the system.

Low-hanging fruit

Try some simple, small things, like tracking down a minor bug that has a very direct correlation to an event you can start hunting from (e.g., a GUI activity). Start with a small, repeatable, low-risk fault report, rather than a meaty intermittent nightmare.

Inspect the code

Run the codebase through some code validators (like *Lint*, *Fortify*, *Cppcheck*, *FxCop*, *ReSharper*, or the like). Look to see if compiler warnings have been disabled; re-enable them, and fix the messages. This will teach you the code structure and give you a clue about the code quality.

Fixing this kind of thing is often not tricky, but very worthwhile; a great introduction. It often gets you around most of the code quickly. This kind of nonfunctional code change teaches you how things fit together and about what lives where. It gives you a great feel for the diligence of the existing developers, and highlights which parts of the code are the most worrisome and will require extra care.

Thonny: Python IDE for Beginners

Silas S. Brown introduces a new Python IDE.

Those of us following (or just occasionally checking) the Raspberry Pi blog might be aware of the Raspberry Pi foundation's recent approval of Thonny (http://thonny.org) as a Python 3 IDE for beginners, and for good reason. Not only is it a simple all-inclusive setup for Python, a syntax-highlighting editor (with library-method completion) and the Pip package manager (with a GUI front-end); it also includes a variable inspector, 'step over' and 'step into' debugging, and a depiction of how expressions are evaluated and which scope a variable applies to. I don't expect advanced programmers to need such things in Python, but I expect they can indeed be useful to beginners, especially young ones, who will also benefit from the philosophy of keeping a simple initial interface (I know how bewildering it can feel to start an IDE that demonstrates all its functions up-front, as if you've just walked up to an aeroplane's cockpit and don't know where to start). I'm glad to see Thonny provided by default on the Raspberry Pi, and it's also an easy install on Windows, Mac (including some older OS X versions) and GNU/Linux (a downloader/installer shell script is provided that's supposed to work on any distribution).

It should be noted that Thonny is for Python 3 only; Python 2 is not supported. A significant number of older Python texts assume Python 2, and the version difference can confuse beginners (in hindsight it might have been better if they'd called the new version Python Plus or something). But if that's not an issue, I'd recommend Thonny for beginners any day. ■

SILAS S. BROWN
Silas is a partially-sighted Computer Science post-doc in Cambridge who currently works in part-time assistant tuition. He has been an ACCU member since 1994 and can be contacted at ssb22@cam.ac.uk

Navigating a Route (continued)

Study, then act

Study a small piece of code. Critique it. Determine if there are weak spots. Refactor it. Mercilessly. Name variables correctly. Turn sprawling code sections into smaller well-named functions.

A few such exercises will give you a good feel for how malleable the code is and how yielding to fixes and modifications. (I've seen codebases that really fought back against refactoring).

Be wary: writing code is easier than reading it. Many programmers, rather than putting in the effort to read and understand existing code, prefer to say "it's ugly" and rewrite it. This certainly helps them get a deep understanding of the code, but at the expense of lots of unnecessary code churn, wasted time, and in all likelihood, new bugs.

Test first

Look at the tests. Work out how to add a new unit test, and how to add a new test file to the suite. How do the tests get run?

A great trick is to try adding a single, one-line, failing test. Does the test suite immediately fail? This *smoke test* proves that the tests are not actively being ignored.

Do the tests serve to illustrate how each component works? Do they illustrate the interface points well?

Housekeeping

Do some spit-and-polish on the user interface. Make some simple UI improvements that don't affect core functionality, but do make the app more pleasant to use.

Tidy the source files: correct the directory hierarchy. Make it match the organisation in the IDE or project files.

Document what you find

Does the code have any kind of top-level *README* documentation file explaining how to start working on it? If not, create one and include the things that you have learned so far.

Ask one of the more experienced programmers to review it. This will show how correct your knowledge is, and also help future newbies.

As you gain understanding of the system, maintain a layer diagram of the main sections of code. Keep it up-to-date as you learn. Do you discover that the system is well layered, with clear interfaces between each layer and no unnecessary coupling? Or do you find the sections of code are needlessly interconnected? Look for ways of introducing interfaces to bring about separation without changing the existing functionality.

If there are no architectural descriptions so far, yours can serve as the documentation that will lead the new recruit into the system.

Conclusion

The more you exercise, the less pain you feel and the greater the benefit you receive. Coding is no different. The more you work on new codebases, the more you are able to pick up new code effectively. ■

Questions

- Do you often enter new codebases? Do you find it easy to work your way around unfamiliar code? What are the common tools you use to investigate a project? What tools can you add to this arsenal?
- Describe some strategies for adding new code to a system you don't understand fully yet. How can you put a firewall around the existing code to protect it (and you)?
- How can you make code easier for a new recruit to understand? What should you do now to improve the state of your current project?
- Does the likely time you will spend working on the code in the future affect the effort and manner in which you learn existing code? Are you more likely to make a 'quick and dirty' fix to code that you will no longer have to maintain, even though others will have to later on? Is this appropriate?

Notes

[1] A single, automatic build step means your build can be placed into a CI harness and run automatically.

Share and Share Alike

Baron M has learned another game.

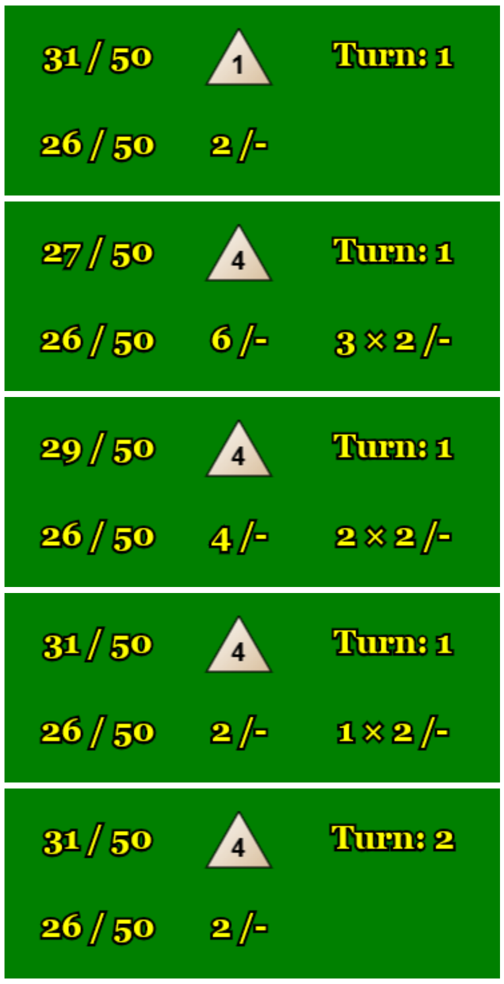
Sir R----- my fine fellow! Come join me in quenching this summer eve’s thirst with a tankard of cold ale! Might I presume that your thirst for wager is as pressing as that for refreshment?

I am gladdened to hear it Sir! Gladdened to hear it indeed!

This day’s sweltering heat has put me in mind of the time that I found myself temporarily misplaced in the great Caloris rainforest on Mercury. I had been escorting the Velikovsky expedition, which had secured the patronage of the Russian Imperial court for its mission to locate the source of the Amazon, and on one particularly close evening our encampment was attacked by a band of Salamanders which, unlike their diminutive Earthly cousins, stood some eight feet tall and wielded vicious looking barbed spears.

Naturally, I leapt into action, dispatching two of their number with my trusty rapier ere they realised that I was there. The vigour with which I prosecuted my attack sufficiently startled them that they took to their heels and fled. I pursued them some distance in their rout and, when I eventually returned to the camp, found it quite abandoned.

In my long search for my fellow explorers, I chanced upon the very spring that was the object of their quest, close to which was a small village in which I determined that I should await their arrival. The inhabitants were of a most egalitarian disposition, happily dividing the fruits of their labours between their neighbours; the hunters their catch, the gatherers their pickings and so on and so forth.



To instil this instinct to share and share alike in the minds of their children they would play a curious dice game with them, which I propose that we employ for our wager.

Here I shall set two coins from my purse upon the table to begin. You shall then cast this four sided die and I shall add to those coins the number that you throw. I shall then divide up the pile of coins into as many piles of equal numbers of coins as I may; if there were six I should make of them three piles of two coins apiece, if there were five then I could make just one pile of five. Of these piles you may keep one for the table and I shall have back the rest for myself. We shall then start again with the pile of coins left upon the table.

The game shall last sixteen such turns and cost you a mere three coins and fifty cents to play.

When I told that loathsome student, whose presence it seems that I am incapable of entirely escaping, of the rules of this wager he paid them no mind whatsoever but instead set to lamenting that the path to his local market had been chained off; why he should not think to take another is quite beyond me!

But enough of his petty grumblings! Come refill your tankard whilst you decide whether or not this wager is to your taste! ■

BARON M

In the service of the Russian military the Baron has travelled widely in this world, and many others for that matter, defending the honour and the interests of the Empress of Russia. He is renowned for his bravery, his scrupulous honesty and his fondness for a wager.

Baron M

Courtesy of www.thusspakeak.com

A Glint of Ruby

Pete Cordell shares his experiences with learning a new scripting language.

I mainly program in C++. I also use C# for Windows GUI programming, PHP for web work and a scripting language for odd tasks such as analysing text files or running tests.

I have been using Perl in the scripting language role. However, even I had to admit that that was getting long in the tooth and it would be good for my skill set to move onto something more modern. The logical choice was Python. I'll admit that I never really got on with Python. The sorts of things I wanted to do often seemed harder to do in Python than in Perl. But Python seemed to be the way things should be done, so I decided to persevere. Then, by chance, I got involved in a project with someone who was using Ruby [1]. After an initial 'whatever', it has been love at first sight ever since!

I tell you this to put the rest of the article into context. I'm not an experienced Ruby programmer, and like an infatuated teenager, I may well be blind to any warts the language might have.

There are many Ruby tutorials online, so I don't intend to teach you Ruby here. Instead, my aim is to help you recognise a Ruby program when you see one, try to pique your interest in learning more about Ruby, and cover a few things that might be intriguing to a C++ (and possibly Python) programmer not familiar with Ruby.

You may already have Ruby on your system if you are using Linux. If not, you should find a suitable download option at [2]. In addition to providing the ability to run Ruby programs on your system, you should also find an interactive Ruby shell called **irb**. This is very handy for experimenting with Ruby, and you may wish to use it to play around with some of the examples below (most of which you should find at [3]).

A contrived example

The obligatory "Hello, World!" program looks as follows in Ruby:

```
puts "Hello, World!"
```

It's always a good sign when a scripting language allows "Hello, World!" to be a one-liner, but it doesn't tell us much about the language. So, in the spirit of feature creep, let's look at Listing 1.

Listing 1

```
# A class to say Hello
class Greeter
  def initialize( who )
    @who = who
  end
  def greet
    if @who =~ /^\s*PETE\s*$/i
      puts "Hello author"
    else
      named_greeting
    end
  end
  private def named_greeting
    puts "Hello #{@who}"
  end
end
greeter = Greeter.new "reader"
greeter.greet
Greeter.new( " Pete " ).greet
```

Working through the example, comments start with the **#** character and continue to the end of the line.

Class names (i.e. **Greeter**) must begin with a capital letter, and, by convention, use PascalCase.

Chunks of code (a 'block' has a special meaning in Ruby so I'm avoiding its usage here) typically start with a keyword such as **class**, **def**, **if**, **while** and continue until a matching **end** keyword.

Indenting is not particularly important in Ruby. The Ruby style guide specifies two spaces per tab. I'm yet to get comfortable with that, but I've adopted it here as it's more idiomatic, and better for presentation in a print publication.

The **def** keyword defines a method. Method names typically are all lower-case and sub-words are underscore separated. The last character can also be one of **?**, **!** or **=**. The **initialize** method is analogous to a constructor in C++, and is called when a new instance of an object is created.

The scope of variables in Ruby is indicated by an optional prefixed character. Without a prefix, a variable is local to the chunk it is defined in. Variables prefixed by a single **@** character have object instance scope. Variables prefixed by **@@** have class scope, similar to C++ **static** class variables. Variables prefixed by **$** are global. So, the line **@who = who** translated to Python would look like **self.who = who**. Variables names must begin with a lower-case letter or underscore, and, by convention, are all lower-case with sub-words separated by underscores (e.g. **my_variable**). (Constants on the other hand must start with an upper-case letter, and are usually all upper-case along with underscore separators, e.g. **MY_CONSTANT**.)

Moving onto the **greet** method, we get our first glimpse of how minimalist and token free Ruby can be. There are no brackets around the **if** clause and there is no delimiter such as **then** or **:** to explicitly mark the end of the condition. One of the goals of the Ruby designer (Yukihiro Matsumoto, or Matz to his friends) was to make programming faster and easier. One aspect of this is to make the interpreter work harder in order to make life for the programmer easier. Typically a Ruby expression ends at the end of a line, unless there is some kind of binary operator, or a **,** to suggest that there is more on the next line. Sometimes additional brackets are needed for disambiguation (and semi-colons can be used to terminate expressions), but the preferred Ruby style is to avoid them if possible. This can take a bit of getting used to, but after a while the result looks more narrative like than typical C++ programs.

The condition of the **if** expression shows that regular expressions have first-class support in Ruby. Those with a Perl background will find this syntax familiar.

The **puts** method call should be familiar to C++ programmers, although you may be excused for having forgotten about it. **puts** prints its arguments to standard output, followed by a newline sequence. If you don't want the newline characters, use **print** instead.

PETE CORDELL
Pete Cordell started with V = IR many decades ago and has been slowly working up the stack ever since. Pete runs his own company, selling tools to make using XML in C++ easier. Pete can be reached at accu@codalogic.com.

Moving to the **else** clause we have a call to the **named_greeting** method. Unlike in Python, there is no need to prefix a call to another method in the same object with **self.**.

The **named_greeting** method definition itself is preceded by the **private** method, showing Ruby gives you the ability to control access to object methods. Note that **private** isn't a keyword like in C++, but a method call. There's some Ruby subtlety here which is too detailed to go into in this article. Suffice to say that the use of **private** on the same line as a **def** will make only that method private (similar to Java and C#'s usage), whereas **private** on its own line (a call to **private** with no arguments) will make all following methods private (similar to C++'s usage).

The **puts** line in **named_greeting** shows Ruby's syntax for string interpolation. Many scripting languages support string interpolation, but Ruby takes this to the max. The code between the opening **#{** and closing **}** can be any expression and is not limited to just variables. It can include operators, method calls, and even **if** statements. If the result of the expression is not a string, then the interpreter will automatically convert it to one.

The class definition described creates a runtime object called **Greeter**. That object includes a method called **new**. When the **new** method is called on the **Greeter** object, it creates a new object that conforms to the class definition. It then calls the newly created object's **initialize** method with the arguments given in the **new** method call. Note again that there are no brackets around the arguments to the **new** method call.

In the example, we assign a reference to the new object to the **greeter** variable. Once we have a reference to an object instance, we can invoke methods on it as shown by the expression **greeter.greet**.

As you might expect, Ruby is garbage collected so there's no need to clean up any objects that have been created.

Hopefully it won't be a surprise to you that the output of the program is:

```
Hello reader
Hello author
```

Digging for gems

Now you hopefully have a feel for what a Ruby program looks like, let's have a closer look at a few things that might be intriguing to a programmer who only knows C++, and things that are more unique to Ruby.

Objects everywhere

Everything in Ruby is an object. There are no primitive, machine word level types such as **char**, **int** and **float** like you find in C++. Consequently, you can call methods directly on explicit values such as floating-point numbers. For example, **1.5.round** evaluates to **2**.

All classes are open to extension in Ruby and are similar to C# partial classes. For example, you can augment the **Float** class with your own methods (although this isn't a recommendation to do it). The following program outputs **4.5**:

```
class Float
  def triple
    3 * self
  end
end
puts 1.5.triple
```

It may look like the usage of **puts** above is a free-standing function. But it is actually a member of the **Object** class from which all objects derive. The other 'fudge' is that the outer-most level of execution takes place within the scope of an automatically defined, but largely hidden, object called **main**.

Types and what's true

In addition to numbers, strings and the regular expressions already mentioned, Ruby also supports arrays and hashes using the handy syntax you'd expect from a scripting language. Additionally, Ruby has a range type which captures a start value and an end value within a single object. I'll touch on ranges in a bit, but here's some examples of the syntax for these types:

```
my_array = [ 1, 2, 3, "Four", 5 ]
my_hash = { "author" => "Pete",
            "reader" => "you" }
my_range = 0..10
```

Ruby will automatically convert between numbers of different types (for example, integer to floating point number) as the operations require. But unlike some languages, outside of string interpolation, it won't automatically convert to and from numerical types and strings.

As with Python, but not with Perl, Ruby has a **Bignum** type that can store integers of any size. If integer operations get too big to fit within a native machine sized integer, they will be transparently converted to a **Bignum**. This is shown in the following code, where ****** is the 'to the power of' operator, and the output is shown in the comments:

```
puts 2**4           # 16
puts (2**4).class   # Fixnum
puts 2**70          # 1180591620717411303424
puts (2**70).class  # Bignum
puts (2**70/
      2**60).class  # Fixnum
```

Then there is **true**, **false** and **nil**. **nil** is equivalent to C#'s **null** and Python's **None**. Only **false** and **nil** are treated as false in conditional expressions. Everything else is treated as true, including **0**, empty strings and empty arrays. Those familiar with Python and Perl might find the latter surprising, but it does seem to fit in with the language quite well. If nothing else, it's easy to remember!

The block

One of the more unique features of Ruby is the concept of a 'block'. Similar to lambdas, any method can be given a block which it can call with a set of parameters, and receive a result back. The 'block' follows a method call, and is either contained between opening and closing braces, or the **do** and **end** keywords. The former syntax is typically used for one-line (or even in-line) blocks, and the latter for multi-line blocks. The arguments to the block are delimited by a pair of **|** pipe characters, and the value of the last expression executed in the block is returned to the calling method. For example, using the variables set up previously:

```
my_array.each { |x| print "_#{x}" }
puts  # Put new line at end of above
my_hash.each do |role, name|
  if role == "reader"
    puts "#{name}, Thanks for reading"
  end
end
```

Outputs:

```
_1_2_3_Four_5
you, Thanks for reading
```

Blocks are not limited to iterating through arrays and hashes.

If the **File.open** method is given a block, it will close the opened file on return from the block. As such, this usage scenario is similar to C#'s **using** and Python's **with** constructs.

In combination with suitable methods, blocks are often used as an alternative to conventional counted **for** loops. In place of C++ like:

```
for (int i=0; i<10; ++i)
  std::cout << "_" << i;
```

you can do any of these (wherein the last one is an example of the range type I mentioned earlier):

```
10.times { |x| print "_#{x}" }
0.upto(9) { |x| print "_#{x}" }
(0..9).each { |x| print "_#{x}" }
```

Why I Avoid PHP

Silas S. Brown shares a war story.

Readers of the *CVu* 'Members' column will notice that for some time ACCU has been seeking a volunteer to update a website that was done in Xaraya, a PHP framework. This has reminded me of my own (negative) experience of a project using the language. My experience has made me wary of volunteering even though I have no reason to believe I would encounter similar issues. I'm writing this, not to put anybody off, but as a starting point for discussion. Is there something wrong with my attitude? Have you had experiences – bad or good – with a language outside its technical aspects that have affected your willingness to use it again? (This article also has some relevance to Francis Glassborow's article in *CVu* 29.3, asking us to discuss other languages, although I fear it's not the type of response he wanted.)

I have some friends who started small businesses. One of them wanted a website (with a database and various bits of business logic and so on), and outsourced it to an offshore development company. They implemented it using Laravel (a PHP framework). My friend then said they weren't happy with the result and needed it to be fixed in a week. And said could they pay me to look at it, as I'm supposedly a good Computer Scientist and PHP is supposedly an 'easy' language, so surely I could fix it in a week, couldn't I?

I said I'll take a look. Someone at that company showed me some of the source code on his laptop. It seemed to have meaningful variable names and to be reasonably formatted, so I said it looks good so far, but I'll need to browse the whole thing myself. Could I sign an NDA they said. Which I did (as, in my experience, if you have to sign a non-disclosure agreement to see something, then the chances are it's not going to be high-quality enough for disclosure to be desirable in any case), and then they gave me the root password to the virtual machine they were hosting it on.

So I started reading through it (I always like to look through any new codebase, to see what's where), and reading through it, and reading through it, and.... And then I had the bright idea of doing a line count. 830,000 lines of PHP, 81,000 lines of CSS, and nearly 300,000 lines of Javascript.

I was shocked. How could I even finish reading this thing on time, let alone making changes? And how did a simple small-business website become 1.2 million lines of code?

Well I could see how the CSS blew up. I thought 'responsive Web design' was supposed to mean using sensible CSS rules to specify how the elements on a page are positioned with respect to one another, using

SILAS S. BROWN

Silas is a partially-sighted Computer Science post-doc in Cambridge who currently works in part-time assistant tuition. He has been an ACCU member since 1994 and can be contacted at ssb22@cam.ac.uk

A Glint of Ruby (continued)

Blocks, and the fact that most methods return something 'useful', opens up the way to a more functional coding style. Among many functional-like methods, Ruby supports `map`, `select` and `reduce` on arrays. Using these, if you wanted to know the sum of the odd squared numbers, for the numbers 0 to 100, you could do:

```
puts (0..100).map { |x| x * x }
  .select { |x| x.odd? }
  .reduce { |acc,x| acc + x }
```

Why Ruby uses `select` rather than the more traditional `filter`, I'm not sure. If it bothers you, though, you can do `class Array; alias filter select; end` and use `filter` in your code instead.

One thing that perhaps should be said here is that I haven't been able to see how to make an expression like the above 'lazy' in the same way that Haskell does. Each method call creates a new array before the next method acts on it, rather than just acting on iterators. This won't cause problems in many cases, but does prevent Haskell-like solutions of the form 'for all positive integers…'.

There's more...

There's much more to Ruby than I've been able to cover here. Ruby has parallel assignment, slices, exceptions, threads and coroutines.

It also has an extensive library of third-party code in the form of RubyGems [4], and a dependency management system called Bundler [5] that ensures that the correct versions of the Gems are installed for your program.

Finding out more

If you want to explore further, there are numerous Ruby tutorials on-line (e.g. [6]). I have found *The Ruby Programming Language* book [7] to be very good. This is a depth-first look at the language that assumes you already know a bit about programming. It feels as close to a language definition as you can get without having to be a language lawyer. The Ruby documentation [8] is good, but I find it hard to navigate. Hence, I tend to use Google as it's front-end! Googling will also bring you to Stack Overflow. Perhaps because Ruby is slightly less common than other languages, the answers for Ruby seem to be of a higher quality than you might find for other languages. The `irb` interactive Ruby shell is also a fun way to poke around to test your understanding.

Conclusion

I hope I've given you enough to be able to recognise a Ruby program. Possibly you're now interested in finding out a little more information. I can understand that some people might find the various naming constraints difficult to get over, but in my case I had already adopted similar rules, and so wasn't bothered. My biggest gripe is that if your code changes a constant, the Ruby interpreter will only generate a warning rather than an error. I console myself that, for the sorts of things I use Ruby for, I don't need a lot of constants. Those wanting to make more extensive use of Ruby might consider this more of an issue.

Overall, I've found code solving similar problems is simpler and clearer in Ruby than in other languages. As such, it has exceeded my expectations and I hope it will for you too. ■

References

[1] https://www.ruby-lang.org/en/
[2] https://www.ruby-lang.org/en/downloads/
[3] https://github.com/codalogic/accu-ruby
[4] https://www.ruby-lang.org/en/libraries/
[5] http://bundler.io/
[6] https://www.ruby-lang.org/en/documentation/
[7] Flanagan, D & Yukihiro Matsumoto, Y (2008), *The Ruby Programming Language*, O'Reilly.
[8] http://ruby-doc.org/core-2.4.1/

Standards Report

Roger Orr reports from the latest C++ meeting.

Our usual correspondent, Jonathan Wakely, is an expectant father at the time of writing and so I have offered to write a report instead.

The last C++ committee meeting was in Toronto in July and was, as usual, a busy week. There were around 120 people present for the meetings, which were held at the University of Toronto and also sponsored by the Fields Institute, Waterfront International, Codeplay, IBM, Google, and Christie Digital. Nine national bodies were represented: Canada, Finland, France, Poland, Russia, Spain, Switzerland, UK, and US.

The ISO voting on the draft International Standard for the 5th edition of C++ (colloquially referred to as 'C++17') was still in progress during the meeting – the voting period actually ends on Sept 3rd. All being well it is

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

Why I Avoid PHP (continued)

measurements relative to the font and screen size but without depending on exact numbers of pixels. I *didn't* think it meant 'create 52 particular screen sizes and apply a separate set of pixel-driven stylesheet rules to each, with lots of duplicate code'. Oh and some of the CSS and Javascript was 'minimised' (i.e. obfuscated) so it wasn't at all easy to debug.

So the first thing I did was to disable 51 out of the 52 screen sizes, and improve the remaining rules so that they worked properly without having to special-case anything (I think that's called 'fluid design'). But then they complained I was breaking more than I was fixing, and no wonder: there was far too much coupling between the CSS, the Javascript, and the PHP; I couldn't just change something and expect nothing else to break.

I don't know which version of Laravel they had used. It certainly wasn't the latest version, and the latest Laravel documentation didn't help me. Of course the company had no idea what the outsourcers had done, and the outsourcers were no longer available for questioning. And nobody had thought to save a copy of the documentation that corresponded to the old version of Laravel in use, or at least make a note of which version that was. Comments had been removed and things had been customised. And the more I tried to make sense of the code, the less sense it made: I kept finding components that seemed to be completely irrelevant. Does that outsourcing team simply dump their entire code-base into every project, switching parts of it on and off as necessary? And yet I couldn't find any obvious place I could use as a starting point to tell me which parts of the code were actually relevant. Did they use some kind of front-end or IDE to edit all this? I only had Emacs. They wouldn't tell me what else to use.

When I voiced my complaints, their first reaction was 'we do things differently in our country', as if I'm a bigot to say a design is bad when it just happens to have been made in a different country. (Should I go out of my way to find that country's top designers and ask them to back me up? Or find a few examples of bad design from my own country just to make it even?)

"Why can't you just change the layout?" they said. Surely it's as simple as that isn't it: just change the layout. You're a computer scientist, remember? This should be easy for you.

I sent them the words of Rupert Gould, as portrayed by Jeremy Irons in Charles Sturridge's 1999/2000 television adaptation of *Longitude*, when Gould was trying to restore John Harrison's early prototype maritime clock:

> I'm going crazy! No don't worry, not that crazy. It's this machine. There's not a straight line in it. It's layer upon layer of corrections, each one fitting on top of the other. Whenever he came across a problem, instead of going back to the beginning, he'd add another level of complexity. Springs, working against levers, working against other springs and other levers, it's madness! This man is born of a refusal to be wrong! He couldn't just say "I've made a mistake"; he'd say, "I'll add something else and then it won't BE a mistake."

(If you want to find that scene, try looking just before the 30-minute mark, but there might be differences between the original UK release and the cut-down US one. I'd like to be able to show the clip on demand in any conversation. I wonder if the scriptwriters realised how salient that scene is for some software-development situations.)

Still they kept insisting that I was a good enough computer scientist to be able to fix it easily. I was tempted to say "OK, I'm not really a computer scientist, all my qualifications are fake" just to get out of that. I suggested they let me re-write the whole thing from scratch to an internal design that I can understand. They said I wouldn't be able to do a rewrite in time. I said that's probably true, but the chances of being able to do a rewrite in time are actually greater than the chances of being able to fix the original in time. They didn't buy it.

And then I realised I did actually have a trick up my sleeve. My Web Adjuster! Never mind trying to fix the PHP: just bolt my Web Adjuster on the front, and set it to make all the changes the company wanted as regular-expression substitutions on the HTML and CSS going out to the browser. Yes it would make me guilty of 'adding another level of complexity' myself, but it would at least buy time and then I could suggest a rewrite later.

But then I made the mistake of making certain substitutions global instead of restricting them to particular pages, and that meant they had unwanted side-effects across the rest of the site, reinforcing the idea that I was creating more problems than I was solving. That would have been a simple change to fix, but they said "forget it – we've had a meeting and decided to abandon the launch, and we're getting a UK company to rewrite it in Wordpress". They were nice enough to pay me something anyway, although it was only about 35% of what they'd originally said. (I said OK, although just to prove a point I did then put in the fix for that 'last straw' problem. Their style of management involved looking at 'how many pages you've got through so far', which didn't particularly help them understand the concept of a global change.)

This has all left me with a rather bad impression of PHP. While I'm sure it is indeed possible to write beautiful code in PHP, it seems the barrier to entry is so low that much (perhaps most?) existing production PHP code is bloated, badly-designed beginners' 'cruft' that shouldn't be anywhere near a serious commercial business. That means there's a high probability that fixing an existing PHP project will be frustrating, and I certainly can't recommend the idea to students who want marketable experience, since I wouldn't wish a PHP career on my worst enemy. Perhaps Rasmus Lerdorf could tell me why it's not his language itself that's the problem, but the codebase? Just don't go there. ■

hoped that all the national bodies will vote 'Yes' and we'll soon be able to report that we do in fact have C++17.

In the meantime, work is progressing on the next edition of the standard – provisionally named C++20. Somewhat to my surprise, we ended the week by adding a sizeable feature to the next standard: the plenary session voted by a large majority in support of a paper adding **concepts** into the C++ working paper by merging in much of the Concepts Technical Specification (TS). While this has been on the agenda for C++20 (after we failed to achieve consensus for doing so in C++17) I was not expecting this to happen so quickly – and with such a high level of support.

This vote had been preceded by a long discussion on Tuesday afternoon, resulting in accepting a number of relatively minor changes to the existing Concepts TS. These include:

- Overloading on constrained functions restricted to named concepts. (The existing matching on the expressions used in the requires clause was found to be problematic in some cases.)
- Adopting a **single** concept definition syntax. The new syntax moves away from the alternatives in the TS of function template and variable template syntax to a slightly simpler form, which looks like a variable template but without the (unnecessary) specification of the `bool` type. It is also a grammar term in its own right, which means future enhancements to the syntax are possible without causing interference with other types of template. A simple example of the new syntax defining a concept for 'integer sized' types might be:

```
template <typename T>
concept int_sized = (sizeof(T) == sizeof(int));
```

This discussion in turn was followed by an evening session, hosted by the Evolution working group, which primarily focussed on the so-called 'abbreviated' syntaxes, which have proved quite controversial. During this session, two polls were taken about this specific issue. There was a very small majority in favour of merging everything achieved so far into the working paper and a strong majority (4:1) in favour of moving everything *except* some of the abbreviated syntaxes into the working paper. This does not preclude adding abbreviated syntax later – there was almost unanimous support for encouraging further work on this.

Andrew Sutton (the Concepts project editor) and Richard Smith (the C++ standard project editor) worked extremely hard – and accurately – to produce draft wording for this merge for the Core working group to start reviewing after lunch on Wednesday. We spent most of two days in CWG on wording review of the paper and also made a few small drive-by fixes of the wording and completed the review in time for a vote on the paper [1] on Saturday.

So, what have we got? Given a concept, such as `int_sized` above, we can write:

```
template <typename T> requires
  int_sized<T> bool foo(T t);
```

or, equivalently, we can write this as:

```
template <int_sized T> bool foo(T t);
```

The function `foo` will only be visible in the overload set when the *constraint* is satisfied: in this case `sizeof(T) == sizeof(int)`.

The Concepts TS provides for two additional syntaxes that are as yet *not* merged, they are the *template introduction* syntax:

```
int_sized{T}
bool foo(T t);
```

and the *abbreviated function template* syntax:

```
bool foo(int_sized t);
```

Note that there are a variety of different terms being bandied about for these two syntaxes: for this report, I've stayed with the terms used in the TS itself.) The Ranges TS, which is based on the Concepts TS, does not use either of these syntaxes in its specification so any decisions about merging this into the working paper are not dependent upon making progress with the abbreviated forms. (There are a couple of small edits that will be needed before merging, but nothing major.)

This was probably the biggest news of the meeting, but there was a lot going on with other technical specifications as well.

Three technical specifications are now completed, and will be moved to publication once the final changes from the week are made and reviewed. (So there is, at present, no final document to link to: for the latest working papers see [2], [3] and [4].)

These are the Coroutine TS, the Ranges TS, and the Networking TS.

The Coroutines TS is based on work by Microsoft, with considerable design input from the concurrency working group, and has recently also been implemented on Clang trunk.

The Ranges TS is a 'conceptised' version of the STL algorithms, with additional features included such as support for ranges (!) – that is, objects with a `begin` iterator and an `end` – and support for projections to enable on-the-fly data transformations. While specified in terms of the Concepts TS, there is an experimental implementation without them [5].

The Networking TS is heavily based on Boost.Asio, so has a long track record and much user experience. We're now getting close to having a standardised network library in C++, which will remove an embarrassment in today's connected world!

The intention is for experience to be gained with these three – both in implementing them and also in using them. For example, Gor Nishanov (the project editor for the Coroutine TS) is keen for someone other than him to implement the Coroutine TS in a compiler to validate that the wording is sufficiently precise. The wider the variety of compilers offering each TS, and of users trying them out, the more sure we will be that the design is ready to add into the main C++ standard.

Finally, the Modules TS is now ready for voting [6], and the various national bodies will shortly receive the draft for review. I personally hope we get the TS published soon, as this will give people a chance to work on different implementations and experiment with using it. We can then see what the benefits and difficulties really are and how much it delivers of what has been anticipated!

Those who attended this year's conference and enjoyed Herb Sutter's closing keynote may be interested to know that he presented a paper about meta classes [7] at the Toronto meeting in an evening session. (Those who missed the keynote can now watch the video – this was embargoed by Herb until after the Toronto meeting [8].) There was a lot of interest in Herb's direction, and strong encouragement to address this level of metaprogramming. I anticipate we'll see further papers in this area, but it will need other facilities – such as reflection – to be included in the working paper before something like this could be considered for inclusion.

The next WG21 meeting will be in Albuquerque, New Mexico, in November. This is incidentally where the ACCU Chair Bob Schmidt lives, so there might be opportunity for an informal ACCU get-together sometime during the week.

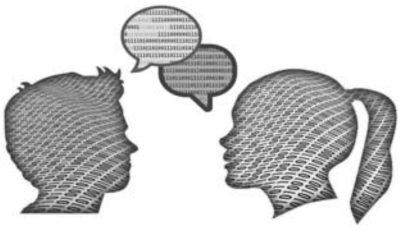
I note that WG14, the C working group, is also meeting in Albuquerque; it is the week before the C++ meeting. This is intended to make travel arrangements simpler for the people who attend both meetings. They are working on a C17 version (this will be a Technical Corrigendum, i.e. containing fixes for bugs in the C11 standard) and also looking for a major revision in the future.

References

[1] Wording Paper, C++ extensions for Concepts, http://wg21.link/p0734
[2] Working Paper for the Coroutine TS: http://wg21.link/n4649
[3] Working Paper for the Ranges TS: http://wg21.link/n4671
[4] Working Paper for the Networking TS: http://wg21.link/n4656
[5] Experimental range library for C++11/14/17: https://github.com/ericniebler/range-v3
[6] Modules PDTS: http://wg21.link/n4681
[7] Metaclasses: http://wg21.link/p0707
[8] https://www.youtube.com/watch?v=6nsyX37nsRs

Code Critique Competition 107

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 learning some C++ by writing a simple date class. The code compiles without warnings but I've made a mistake somewhere as the test program doesn't always produce what I expect.
>
> ```
> > testdate
> Enter start date (YYYY-MM-DD): 2017-06-01
> Enter adjustment (YYYY-MM-DD): 0000-01-30
> Adjusted Date: 2017-07-31
> >testdate
> Enter start date (YYYY-MM-DD): 2017-02-01
> Enter adjustment (YYYY-MM-DD): 0001-01-09
> Adjusted Date: 2018-03-10
> >testdate
> Enter start date (YYYY-MM-DD): 2017-03-04
> Enter adjustment (YYYY-MM-DD): 0001-00-30
> Adjusted Date: 2017-04-03
> ```
>
> That last one ought to be 2018-04-04, but I can't see what I'm doing wrong.

Please can you help the programmer find his bug – and suggest some possible improvements to the program!

- Listing 1 contains `date.h`
- Listing 2 contains `date.cpp`
- Listing 3 contains `testdate.cpp`

Listing 1

```
// A start of a basic date class in C++.
#pragma once

class Date
{
  int year;
  int month;
  int day;

public:
  void readDate();
  void printDate();
  void addDate(Date lhs, Date rhs);
  bool leapYear();
};
```

Listing 2

```
#include "date.h"
#include <iostream>
using namespace std;

// Read using YYYY-MM-DD format
void Date::readDate()
{
  cin >> year;
  cin.get();
  cin >> month;
  cin.get();
  cin >> day;
}
// Print using YYYY-MM-DD format
void Date::printDate()
{
  cout << "Date: " << year << '-' <<
    month/10 << month%10 << '-' <<
    day/10 << day %10;
}
void Date::addDate(Date lhs, Date rhs)
{
  year = lhs.year + rhs.year;
  month = lhs.month + rhs.month;
  day = lhs.day + rhs.day;
  // first pass at the day -- no months
  // are over 31 days
  if (day > 31)
  {
    day -= 31;
    month = month + 1;
    if (month > 12)
    {
      year += 1;
      month -= 12;
    }
  }
  // normalise the month
  if (month > 12)
  {
    year += 1;
    month -= 12;
  }
  // now check for the shorter months
  int days_in_month = 31;
  switch (month)
  {
    default: return; // done 31 earlier
    case 2: // Feb
      days_in_month = 28 + leapYear()?1:0;
      break;
    case 4:  // Apr
    case 6:  // Jun
    case 9:  // Sep
    case 11: // Nov
      days_in_month = 30;
  }
```

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

Listing 2 (cont'd)

```
  if (day > days_in_month)
  {
    day -= days_in_month;
    month += 1;
    if (month > 12)
    {
      month -= 12;
      year += 1;
    }
  }
}
bool Date::leapYear()
{
  // Every four years, or every four centuries
  if (year % 100 == 0) return year % 400 == 0;
  else return year % 4 == 0;
}
```

Listing 3

```
#include "date.h"
#include <iostream>
using std::cout;

int main()
{
  Date d1, d2, d3;
  cout << "Enter start date (YYYY-MM-DD): ";
  d1.readDate();
  cout << "Enter adjustment (YYYY-MM-DD): ";
  d2.readDate();
  // Add the two dates
  d3.addDate(d1, d2);
  cout << "Adjusted ";
  d3.printDate();
}
```

Critique

Jason Spencer <contact+pih@jasonspencer.org>

I have to admit, I can't reproduce the error. I get 2018-04-03 for the last test. Perhaps if there was more information on OS, compiler and STL used I could get 2017-04-03. I suspect this is Windows (the **>** command prompt) and Visual Studio (the **#pragma once** directive is non-standard and used there more often), but don't have such a dev environment to hand right now. Is it possible your build manager is using an old object file with different date calculation logic? Or that you've compiled one file but pasted a different file? (It happens to all of us.)

I get 2018-04-03 – the reason it is not 2018-04-04 is because your carry logic in **addDate** adds 30 to 4, gets 34 and then subtracts 31 – the remainder is 3, **days_in_month** is set to 30 in the **switch** statement (month was earlier incremented to 4), but that's more than the day, so nothing else happens. There are *coincidentally* 31 days in March, so 2018-04-03 is correct. If you'd added 0000-00-30 to a February date, you'd get the wrong answer. If you were adding across a leap year, you'd also get the wrong answer.

As to why you might get 2017 – I can't see it – there's no decrement in the year and no pointer usage. Perhaps there is some garbled input from the console when reading the adjustment date, and because there is no input validation (of range, **int** parsing or delimiters) the date may be read as year=0, month=1, day=0, ie the adjustment shifted to the right with the 30 left in the buffer (although that would return 2017-04-04 as the adjusted date). This shouldn't happen though – if there is garbage on the input, the first **int** parse will set the fail flag on the stream and the later **int** parsings will return with the variables unchanged (unchanged and therefore uninitialised!).

To test what is being read, put a debugger breakpoint on **readDate** and step through it while monitoring the values, or just print the year, month and day values at the end of the function.

Taking the functions in order, I'd comment thusly:

- **Date::readDate()** should check whether the delimiters are correct. Perhaps some bounds checking on the read day, month and year. If you use the same class to also express a duration then this becomes a problem (you might want to add 20 months, but that would be an invalid date). '-' is not only a delimiter but a valid prefix to a signed **int** and could be consumed by the **int** read, so check bounds. "-1--1--1" is a valid date. Your code would also accept "a-a-a", "abc", "a". There should be a test to see if the stream is valid at the end of the function – to detect EOF, and use **cin::fail()** to check if the integer has been correctly parsed.
- **Date::printDate()** should have a **const** suffix as it doesn't change the object [1]. Rather than print the 1s and 10s separately, presumably to print "01" and not "1" for the first month, consider **setfill('0')** and **setw(2)** stream manipulators from **<iomanip>** instead.
- **Date::addDate(Date lhs, Date rhs)** is almost impenetrable. Rather than implementing the number of days in a month as logic in a switch statement, consider instead a look-up table and also try to avoid magic numbers. Try to make your code as self-documenting as possible. Also, you don't need to pass the objects by mutable (ie non-const) copy. You could try by const reference: **Date::addDate(const Date & lhs, const Date & rhs)**, but then the compiler has to insert possibly superfluous memory reads because lhs and rhs may be the same object (see "pointer aliasing"), and may even be the current object **d1.addDate(d1,d1)**, which will probably corrupt the result. **Date::addDate(const Date lhs, const Date rhs)**, on the other hand, will copy the **Date** objects on to the current stack frame, and may elide (omit) the copy if it sees it's unnecessary (see 'copy elision'). It also means that when implementing **addDate**, the compiler will complain if you accidentally write to lhs or rhs.

  Do we need two arguments to the method? This function is actually doing two things – addition and assignment. Perhaps **Date::addDate(const Date other)**?

  And your **addDate** logic assumes that the adjustment date is positive – it might not be.
- **Date::leapYear()** should have a **const** suffix [1], but I'd also propose a second version that is a static method, so that the programmer can test whether a specific year is a leap year without having to create a dummy **Date** object. The non-static method should call the static method so you don't repeat yourself (see 'DRY principle') and keep the leap year logic in one place. Consider renaming to **isLeapYear()** to make it obvious it is a test.
- In terms of general class design, you really should initialize the year, month and day member **int**s in a default constructor, **Date::Date(): year(0), month(0), day(0) {}**, as C++ doesn't always default initialise built-in data types. In **main()**, try printing d3 before the call to **addDate** to see what I mean.

So now the main problem with this code – a conceptual error. What does it really mean to add two dates? Is the year you are adding 365 or 366 days long? And in another situation, are the 7 months you want to add each 28, 29, 30 or 31 days long? And in what order do you add the year, month and day? The order affects the result. I'd suggest the simplest way to remove the ambiguity is not to add dates at all, but add a duration to a date.

If you want to program more defensively then you could create wrappers for days, months and years before they are passed to **Date** methods:

```
class Days {
  int days;
public:
  explicit Days( int days ): days(days) {}
  operator int() const { return days; }
};
```

Then you could create **Date::add(const Days);** and call **d1.add(Days(1000))** if you want to add 1000 days to **d1**. Do not allow the addition of **`Date`** to **`Date`**. If you also create **`Months`** and **`Years`** wrappers then you can have a **`Date::Date(const Years, const Months, const Days)`** constructor which removes the confusion of US vs ISO ordering:

```
Date d1 ( Years(2017), Months(3), Days(4) )
```

The **`Days`** constructor is marked as explicit so that the compiler doesn't attempt to automatically convert between types.

I suppose you could also create an overloaded **`Date::add(const Years)`** and do **`d1.add(Days(30)); d1.add(Years(1));`** This moves the responsibility of the ordering of the addition on to the **`Date`** class user, but do still be very careful that you do the carry correctly. In fact, perhaps reconsider whether storing three **`int`**s is the best way to represent the date internally – you could store the number of days since a fixed epoch (see 'Unix time', and the 2038 problem).

And rather than using program logic to determine how many days there are in a month, why not try a lookup table stored in a static protected member variable like:

```
static const constexpr struct {
  char name [MONTH_NAME_MAXLEN+1];
  uint8_t days_nonleap_year;
  uint8_t days_leap_year;
  } MONTH_DETAILS [] = {
    { "Jan", 31, 31 },
    { "Feb", 28, 29 },
    { "Mar", 31, 31 },
...
    { "Nov", 30, 30 },
    { "Dec", 31, 31 },
    { "", 0, 0 }
};
```

I've included a human readable month name for convenience. The struct could also be broken out into separate arrays (**`MONTH_NAMES[],DAYS_PER_MONTH_NON_LEAP_YEAR[],DAYS_PER_MONTH_LEAP_YEAR[]`**) to simplify internationalisation.

Having this look up table will also allow you to check input data (in **`readDate`** or equivalent) bounds more easily and uniformly across your code. You could also create a function static:

```
Date::numberOfDaysInMonth(int month)
```

so your class could be used as part of a calendar printer, for example.

To remove various magic values consider using:

```
enum MONTHS { JAN, FEB, MAR, APR, MAY, JUN,
  JUL, AUG, SEP, OCT, NOV, DEC, LAST };
static const constexpr int MONTHS_IN_YEAR = 12;
```

Beware of off-by-one errors when converting the enum to an **`int`**, though (**`JAN`** above is 0, not 1, but would allow you to do **`MONTH_DETAILS[FEB].days_leap_year`**).

Another issue your code has is class design. The **`Date`** class should have one responsibility – to store and manipulate dates, not to read or print them. What if you want to read in a US date format, **`YYYY-DD-MM`** (an almost related story can be found here [2]), or from a string or file?

`readDate` and **`printDate`** should be standalone functions outside of **`Date`**, and then you can do **`readISODate ( std::cin, d1 )`** and **`readUSDate ( std::cin, d1 )`**.

Try to avoid repeating information in method names – instead of **`d1.readDate()`**, why not **`d1.read()`**? Or declare a function outside the class called **`std::istream& operator>> ( std::istream & input_stream, Date & read_date )`**, so you can then do **`cin >> d1`**.

In general have a look at **`Boost:Date`** [3], **`java.util.Date`**, and **`std::chrono`** for design inspiration. And of course they're already known to be working, so use them, when you're not doing this for educational purposes.

In terms of class design, have a look at the books *Clean Code* [4], *Code Complete* [5] and *The Pragmatic Programmer* [6]. All excellent books.

References

[1] https://isocpp.org/wiki/faq/const-correctness#const-member-fns

[2] See root cause here: https://mars.jpl.nasa.gov/msp98/news/mco991110.html

[3] http://www.boost.org/doc/libs/1_61_0/doc/html/date_time.html

[4] *Clean Code* by Robert C. Martin, Prentice Hall, ISBN 0132350882

[5] *Code Complete* by Steve McConnell, Microsoft Press, ISBN 0735619670

[6] *The Pragmatic Programmer* by Andrew Hunt and David Thomas, Addison Wesley, ISBN 020161622X

James Holland <james.holland@babcockinternational.com>

As with many student code examples, it is not entirely clear what the sample code is meant to do. At first glance, it appears that an attempt is being made to add two dates. This doesn't really make sense. I assume what is meant to happen is that the two values of years entered are to be added, the two values of months are to be added and the two values of days are to be added. If the total number of days is greater than the current month, I assume the days and months are to be adjusted accordingly. If this results in the number of months exceeding 12, the months and years are to be adjusted. All this has to be performed while taking into account leap years. Furthermore, can any value be entered for the adjustment year, month and day?

The sample program output provided by the student adds to the confusion. The first two runs are quite easy to work out in one's head and agree with my run of the student's program. The third run is baffling. I do not understand how the program can produce a year of 2017. When I run the program I get 2018 as expected. According to my calculations, 30 days from 4th March is 3rd April, regardless of the year. This value is in agreement with the student's printout and my running of the program. Why the student insists the date should be the 4th of April, I cannot fathom.

Despite these peculiarities, let's have a look at the software to see if there is anything definitely wrong. When adding 38 days to 25th July 2017, for example, the student's program gives 2017-08-32 which is clearly not a valid date. The problem is partly the order in which the student is normalising the days and months, and partly because normalising is not performed often enough and so not completely normalising the date. My solution consists of three **`while`**-loops that ensure complete date normalisation. I will not give further details as they are not specific to C++ and, as the student says, it is the writing of simple C++ classes that is of immediate interest. I have provided a few embellishments to the student's **`Date`** class, described below, that may be of some value.

It is often desirable for programmer-defined objects to behave in a similar way to built-in types. For example, it would be convenient to print the value of a **`Date`** object by using **`<<`** as the student does for text strings. This can be achieved by defining the free function **`operator<<()`** that takes two parameters; a reference to the output stream on which the value of **`Date`** is to be written, and a reference to the **`Date`** object. The function prints the value of year, month and day in a similar way to the student's **`printDate()`** member function. One difference is that I have chosen to format the value of month and day using manipulators. In this way the intent of the code is, I think, more clearly stated. Because **`operator<<()`** needs access to the private members of the **`Date`** class, **`Date`** has to grant **`operator<<()`** permission by declaring **`operator<<()`** as a friend.

Another way to make **`Date`** behave more like built-in types is to allow two **`Date`** objects to be added using a simple **`+`** operator. This is achieved by defining a class member **`operator+()`**. This function takes a reference to the 'right-hand side' **`Date`** object as a single parameter and returns a **`Date`** object that is the sum of the current object plus the one referenced by the parameter. From the attached code, it can be seen that **`operator+()`** returns a **`const`** value. This is to prevent statements such as **`d1 + d2 = d3`** from compiling, as is the case for built-in types.

Finally, I provide a copy constructor and a constructor taking a text string as a parameter. The copy constructor is fairly standard and allows one `Date` object to be constructed from another `Date` object. The constructor taking a string may be considered somewhat unconventional but ensures that the object is initialised and provides a convenient way of specifying the text to prompt the user. Declaring a constructor prevents the compiler from automatically generating a default constructor. This can be considered a good thing in this case as it prevents the user from constructing an uninitialised `Date` object.

Having defined the additional operators and constructors, `Date` objects can be created and manipulated in a more natural way as shown in `main()` of the supplied code.

```
#include <iostream>
#include <iomanip>
using namespace std;
class Date
{
  int year;
  int month;
  int day;
  bool leap_year(int year) const
  {
    if (year % 100 == 0)
      return year % 400 == 0;
    return year % 4 == 0;
  }
  int days_in_month(int year, int month) const
  {
    switch (month)
    {
      case 2:
      return leap_year(year) ? 29 : 28;

      case 4: case 6: case 9: case 11:
      return 30;

      default:
      return 31;
    }
  }
  Date (int new_year, int new_month,
    int new_day)
    : year(new_year), month(new_month),
      day(new_day){}
  public:
  const Date operator+(const Date & rhs)
  {
    int year = this->year + rhs.year;
    int month = this->month + rhs.month;
    int day = this->day + rhs.day;
    while (month > 12)
    {
      month -= 12;
      ++year;
    }
    while (day > days_in_month(year, month))
    {
      day -= days_in_month(year, month);
      ++month;
    }
    while (month > 12)
    {
      month -= 12;
      ++year;
    }
    return {year, month, day};
  }
  Date(string request)
  {
    cout << request;
    cin >> year;
    cin.get();
    cin >> month;
    cin.get();
    cin >> day;
  }
  friend ostream & operator<<(ostream &,
    const Date &);
};
ostream & operator<<(ostream & os,
  const Date & d)
{
  os << d.year << '-'
     << setw(2) << setfill('0') << d.month
     << '-'
     << setw(2) << setfill('0') << d.day;
  return os;
}
int main()
{
  Date d1("Enter start date (YYYY-MM-DD): ");
  Date d2("Enter adjustment (YYYY-MM-DD): ");
  Date d3 = d1 + d2;
  cout << "Adjusted " << d3 << endl;
}
```

Robert Lytton <robert@xmos.com>

It would be helpful to start the code review with first impressions.

1. This looks like something that could be found in a library;
2. The 'WET' principle is being used;
3. The algorithm used by `addDate()` is difficult to perceive;
4. `class Date` is easy to use incorrectly and has a twinge of astonishment.

Let me unpack these impressions and hopefully find the bug in the process.

1. As this is a learning exercise, recreating a solution to a problem is an interesting thing to do. This does not necessarily mean the writer should not make use of lower level libraries during the exercise. More of this later. It should be noted that the cost of not using external libraries includes things such as maintenance & bug fixing; fewer users and tests exercising the code; more tacit knowledge required by people getting on board; Effort to get thing right – the interface.
2. The 'WET' ('Write Every Time') principle can be seen in the code that truncates `months > 12` and increments the year. This fragment is repeated three times during `addDate()`.

   'WET' code is bad for several reasons including:

   a) an opportunity to raise the level of abstraction has probably been missed;
   b) there is more code to compile and to maintain;
   c) if the copies diverge during maintenance, it may be a bug;
   d) there is more code to read and thus hide the essential detail.

   Instead of WET, the 'Don't Repeat Yourself' (DRY) principle should be followed. For `addDate()`, moving the repeated code into a function may be the correct choice.
3. One of the reasons the `addDate()` algorithm is difficult to perceive is the 'WET' code referred to in #2. Refactoring the functionality into `void normaliseMonth(int& month, int& year);` (taking the name from the comment and thus making the comment redundant) makes is easier to follow the intention of the code. Further refactoring could include adding the functions, both adding clarity to the code but keeping to the original intention:

   - `void normaliseDay(int& day, int& month);`
   - `int daysInMonth(int month);`

During such refactorings, the bug may become apparent, or just vanish with the replacement code. Vanishing bugs are not good – they may reappear. Also, we want to understand how the defect happened so we can change our practises to avoid similar defects.

In this case, the bug was in the first block where **`if (day > 31)`** reduces the day by 31 and increments the month. This is incorrect – some months require normalising with values less than 31. Later code normalises correctly, using **`days_in_month`**, but this can't fix the earlier mistake. Could this be a case of #2c above – the WET normalise-day code being fixed in the 2nd but not the 1st occurrence?

The aim of refactoring should be to turn 'no obvious bugs' into 'obviously no bugs' (paraphrasing Tony Hoare). In our case it should be possible to reason with the refactored code and reorder, reduce and simplify – an exercise for the reader.

4. Class design is more than a correct implementation, or even an implementation that is 'obviously correct'. Scott Meyers exhorts "Make Interfaces Easy to Use Correctly and Hard to Use Incorrectly" – let us explore this principle. The class interface consists of the public elements: four methods plus six default methods added by the compiler. The default copy-constructor, move-constructor, copy-initialiser, move-initialiser and destructor all handle the private member data as we would wish.

   However:

   a) The default constructor – does not initialise the member data. To prevent undefined behaviour, the members need to be initialised.

   This may be done with default values in the declaration:

```
int year=0;
int month=0;
int day=0;
```

   or by reusing our **`readDate()`** method:

```
void Date::Date() {readDate();}
```

   or it may be better to specify a constructor that force the user to pick a valid date instead:

```
void Date::Date(int y, int m, int d):
  year{y}, month{m}, day{d} {}
```

   N.B. the use of **`int`** for each parameter in this example makes the constructor easy to misuse (wrong order) – see later.

   b) **`readDate()`** – the implementation is coupled to **`std::cin`** and is not robust. This coupling is hidden from users, is not necessary and reduces usability. An alternative is to have a stream passed in explicitly by the caller:

```
void Date::readDate(std::istream& in);
```

   Or use a non-member function (which calls a Date setter):

```
std::istream& operator>>(std::istream& is,
  Date& d) {
  // read in the values and validate them!
  //...
```

   As the comment suggests, the current version lacks error or sanity checking of input values. Should **`day`** be set to 1000, the class will not work as expected!

   A bonus of using an **`istream`** is that we can using automated unit tests rather than typing!

   c) **`printData()`** – the implementation is coupled to **`std::cout`** and is not **`const`**. The choices are similar to those of **`readDate()`** but the preferred option is to add a **`print()`** method:

```
std::ostream& Date::print(std::ostream &out)
  const;
```

   and a non-member function to call it:

```
std::ostream& operator<<( std::ostream& os,
  const Date& d ) {
  return d.print(os);
}
```

   It should also be noted that printing month and day don't need to use **`/`** & **`%`**.

   d) **`addData()`** – does not follow the 'principle of least astonishment'. This astonishment could be lessened by turning the method into a constructor. An alternative would be to do what other classes do and overload **`+`**:

```
Date& Date::operator+=(const Date &rhs) {
  year += rhs.year;
  //...
}
```

   and add a non-member function too:

```
Date operator+(Date lhs, const Date& rhs) {
  lhs += rhs;
  return lhs;
}
```

   e) **`leapYear()`** – is not an essential part of the interface. This should be removed from the interface by making it private. If it is require later, it may be added at the cost of a recompilation – thus following the 'open close' principle.

We could spend longer digging in deeper with issues such as:

- Constraining values within valid limits using **`unsigned`**, or **`enums`**;
- Using strong types for day, month, year rather than an **`int`** or weak **`typedef`**;
- Checking values entering the class are valid – thus maintaining invariants;
- Using lower-level abstractions to hold and manipulate state on behalf of the class e.g. **`time_t`** & **`time.h`** or **`Boost::DateTime`**;
- White-box unit testing of normalisation paths & combinations of paths;
- The true cost of a class.

But they are not first impressions.

Commentary

First off, an apology: I can't explain the final result I posted in the original critique. While I thought I'd copy-pasted it into the critique from a command prompt, I obviously didn't do this successfully because (a) I cannot reproduce this result and (b) the result is in fact correct and does not demonstrate the bug in the program. I apologise for the confusion this error caused!

I haven't a lot of commentary to add as I think the entries between them cover pretty well all the ground; both the problems with the implementation and the various higher-level issues with the design of the class.

While it was mentioned that you don't need to split the month and day into tens + units to ensure they print correctly, with standard iostreams it is quite painful as you need to:

- save the current state of the fill character and set it to '0'
- set the width to 2 (for each field)
- restore the fill character (avoids future surprises with the ostream...).

For example:

```
// Print using YYYY-MM-DD format
void Date::printDate()
{
  auto orig(cout.fill('0'));
  cout << "Date: " << year << '-' <<
    std::setw(2) << month << '-' <<
    std::setw(2) << day;
  cout.fill(orig);
}
```

It is arguable whether this is better than the original.

The Winner of CC 106

All three entrants did a good job of explaining the problem with the class and suggesting improvements. Pointing out ways in which the design differs from the usual C++ idioms for value types would be particularly useful given the context of someone trying to get more familiar with C++.

On balance, I think Jason's critique was overall the best one, so I have awarded him this month's prize.

Code Critique 106

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

> I want to collect the meals needed for attendees for a one-day event so I'm reading lines of text with the name and a list of the meals needed, and then writing the totals. However, the totals are wrong – but I can't see why:
>
> ```
> > meals
> Roger breakfast lunch
> John lunch dinner
> Peter dinner
>
> Total: 3 breakfast: 3 lunch: 2 dinner: 2
> ```
>
> There should only be 1 breakfast, not 3!

Please can you help the programmer find the bug – and suggest some possible improvements to the program!

- Listing 4 contains `meal.h`
- Listing 5 contains `meals.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 4

```
#pragma once
#include <iosfwd>
#include <sstream>
#include <string>
enum class meal : int
{
   breakfast, lunch, dinner,
};
// Used for name <=> value conversion
struct
{
  meal value;
  std::string name;
} names[] =
{
  { meal::breakfast, "breakfast" },
  { meal::lunch, "lunch" },
  { meal::dinner, "dinner" },
};
std::istream &operator>>(std::istream &is,
  meal &m)
{
  std::string name;
  if (is >> name)
  {
    for (auto p : names)
    {
      if (p.name == name)
        m = p.value;
    }
  }
  return is;
}
```

Listing 4 (cont'd)

```
std::ostream &operator<<(std::ostream &os,
  meal const m)
{
  for (auto p : names)
  {
    if (p.value == m)
      os << p.name;
  }
  return os;
}
// Type-safe operations
constexpr meal operator+(meal a, meal b)
{
  return meal(int(a) + int(b));
}
meal operator+=(meal &a, meal b)
{
  a = a + b;
  return a;
}
constexpr meal operator|(meal a, meal b)
{
  return meal(int(a) | int(b));
}
constexpr meal operator&(meal a, meal b)
{
  return meal(int(a) & int(b));
}
// Check distinctness
static_assert((meal::breakfast | meal::lunch |
 meal::dinner) == (meal::breakfast +
 meal::lunch + meal::dinner), "not distinct");
```

Listing 5

```
#include "meal.h"

#include <iostream>
#include <list>

struct attendee
{
   std::string name;
   meal meals; // set of meals
};

using attendees = std::list<attendee>;

attendees get_attendees(std::istream &is)
{
  attendees result;
  std::string line;
  while (std::getline(is, line))
  {
    std::istringstream iss(line);
    std::string name;
    iss >> name;
    meal meal, meals{};
    while (iss >> meal)
      meals += meal; // add in each meal
    if (is.fail())
      throw std::runtime_error("Input error");
    result.push_back({name, meals});
  }
  return result;
}
```

A New Competition

Francis Glassborow presents a new challenge for *CVu* readers.

Many years ago, when I was editor of *CVu*, I ran a number of programming challenges. The most successful of these was to design a deterministic sort algorithm (i.e. it must terminate so looping through random shuffles and checking if the result was sorted did not qualify). Some readers managed to come up with some truly horrendous algorithms. I suppose that spending so much time working hard to write good efficient and correct code makes the challenge of writing bad, inefficient but correct code a relief.

I was reminded of this by one of the events at the recent ACCU conference where teams clearly enjoyed the challenge of writing code that would compile but with a serious number of constraints such as the twin requirements to keep the character count low whilst including particular keywords or tokens. Some of the entries were tours de force and I am not sure of the adjudicator's decision to rule against a piece of code that compiled because of a bug in the compiler. To my mind that sort of thing should get a bonus.

I thought that it might be fun to run some more coding challenges in *CVu* and I hope that lots of you will take part (not only try the challenge but submit your code to the editor).

I will aim to be imaginative and provide challenges that are nothing like what you have to do in the day job. I do not promise to provide one for every issue but I will do my best.

Challenge 1

Many years ago, a programmer wrote a piece of machine code that ran on two different machines to do two entirely different things. The author had spotted that a specific instruction was valid on both systems but on one it was an unconditional jump and on the other it was effectively a harmless instruction that could be ignored. So the first instruction of his program was that and on one machine the code simply ran from the start; on the other, it jumped to code that was valid on the other machine.

That leads me to your first programming challenge: write a program that will compile both as C and as C++ but will do different things. To keep things simple you need to use a GCC C and C++ compiler. The code must compile for the current versions of C (11) and C++ (14).

The entries will be judged on two criteria.

- One (subjective): how different the output is for the C and C++ versions.
- Two: the ratio of instructions executed in the C version to the overall number of instructions in the original source code. In other words you should strive to have the maximum amount of commonality.

There is a further limitation in that you may only use standard header files.

FRANCIS GLASSBOROW
Since retiring from teaching, Francis has edited C Vu, founded the ACCU conference and represented BSI at the C and C++ ISO committees. He is the author of two books: *You Can Do It!* and *You Can Program in C++*.

Code Critique Competition 107 (continued)

Listing 5 (cont'd)

```
size_t count(attendees a, meal m)
{
  size_t result{};
  for (auto &item : a)
  {
    // Check 'm' present in meals
    if ((item.meals & m) == m)
      ++result;
  }
  return result;
}
int main()
try
{
  auto attendees{ get_attendees( std::cin) };
  std::cout << "Total: " << attendees.size();
  for (auto m : { meal::breakfast,
                  meal::lunch, meal::dinner })
  {
    std::cout << ' ' << m <<  ": " <<
      count(attendees, m);
  }
  std::cout << '\n';
}
catch (std::exception &ex)
{
  std::cout << ex.what() << '\n';
}
```

Bookcase

The latest roundup of book reviews.

If you want to review a book, your first port of call should be the members section of the ACCU website, which contains a list of all of the books currently available. If there is something that you want to review, but can't find on there, just ask. It is possible that we can get hold of it.

After you've made your choice, email me and if the book checks out on my database, you can have it. I will instruct you from there. Remember though, if the book review is such a stinker as to be awarded the most un-glamorous 'not recommended' rating, you are entitled to another book completely free.

Thanks to Pearson and Computer Bookshop for their continued support in providing us with books. Astrid Byro (astrid.byro@gmail.com)

Effective Ruby – Live Lessons

by Sam Phippen, Peter J. Jones and Scott Meyers, published by Addison-Wesley Professional, ISBN: 978-0-13-417537-9

Reviewed by Ian Bruntlett

If you are already familiar with Ruby, skip this paragraph. The first thing you do regarding Ruby is to visit www.ruby-lang.org/en/documentation/ so you can play around with it to see if it suits you. The first detailed book to read about Ruby would have to be *The Ruby Programming Language* by Flanagan and Matsumoto, but this only covers Ruby 1.8 and 1.9. I figure that the Ruby language is changing so much that books and videos are like booster rockets. They can be used to get you off the ground but, once in orbit, you find that you rely on the main Ruby websites: www.ruby-lang.org and ruby-doc.org/.

The bad news. The videos lack a detailed listing of contents and time offsets of subsections. Knowing which lesson is at which point in a video file and how long that lesson is would have saved me a lot of time. During the course of this review, I have created a detailed contents listing. The quality of the video footage of vim being used to edit and run Ruby programmes isn't too good. For example the `[]` operator looks like a square box when viewed in the first chapter. It would have been helpful if the screen was used more intelligently – in particular, showing an example's source code alongside its output would have been particularly helpful. Showing the filename of the example would have made life easier as well (later examples do show the filename and initial directory).

Ruby has more than its fair share of quirks. This is where this set of videos comes in handy. It complements the other books and websites quite well but should be regarded as a booster rocket.

These videos also rely on third party modules for Ruby (colloquially known as 'gems'). You'll be needing to be familiar with Gems and how to install them. On Linux, the command **`man gem`** will get you so far. Then you'll need to visit the website rubygems.org. When you have downloaded the example code of these lessons, there will be files ending in `.rb` – they're the examples and there will be a file called `Gemfile` that lists the external dependencies of the example files.

Another useful resource is **`ri`**. On Ubuntu Linux, the **`ri`** command is installed along with Ruby. However, the information pages for Ruby are in a separate package, rubyn.n.doc, where n.n are the major and minor version numbers of the Ruby interpreter you are using.

The video files are O.K. – there are 5 of them but it would have been very helpful if they'd supplied an overview of their contents, along with time offsets so that people could jump to a particular point when needed.

One of the problems that happen when learning a new programming language on your own is that you don't get to see production code or find out about the 'gotchas' of the language you are learning. From that point of view, these videos are invaluable as it provides both.

I am still learning Ruby so, rather than relying on my opinion of individual items, I'll give a brief overview of each lesson. All languages have 'gotchas' and Ruby is no exception – these videos point out some of Ruby's gotchas.

Lesson 1 – Arrays and Hashes [38 minutes]. This is a reasonably gentle start and not too long. I am not sure why 'Arrays' is in the lesson title – as well as Hashes it covers preserving object encapsulation, using Set instead of Hash and an item on delegation that is useful if you have a class with a hash that you want to partially expose to the users of your class.

Lesson 2 – Seams [37 minutes]. This lesson deals with writing your code to make refactoring easier, later on in life when your system need improving. Some things I slightly disagree with but I cannot dispute the fact that the advice provided here, used rationally, is really important.

Lesson 3 – Testing [42 minutes]. I am using these videos to expand my Ruby knowledge and with this lesson I haven't typed in all the examples and tested them. However, I found this lesson to be particularly useful in showing to me lots of things I wasn't aware of. It answered some unanswered questions that I had about how to test Ruby applications.

Lesson 4 - Enumerables and Callables [1 hour and 25 minutes]. This is a long lesson, but important. Item 17 covers Enumerables and Callables, noteworthy features of Ruby. As mentioned earlier, you really need to read *The Ruby Programming Language* and that is especially true of this lesson. This lesson will take you far further than the use of **`#each`**.

Lesson 5 – The Standard Library [34 minutes]. In general, it is best to use Ruby's Standard Library. It will have less bugs than handwritten code and, if used wisely, will handle platform specific quirks, and may offer better performance. It covers the use of block forms to avoid failing to close files when an exception is raised, a little bit of Net::HTTP and describes how to use the File module's functions for manipulating file and path names.

There is a web page on www.informit.com for these videos – search for "effective ruby livelessons" and you should find it. That page will let you buy a digital download copy of the lessons and has a link to the Ruby source code files for the videos.

Now you have a reasonable idea of what this video course offers. Have fun :)

ACCU Information

Membership news and committee reports

accu

View from the Chair

Bob Schmidt
chair@accu.org

When I started as Chair last year, I took some time to go back and reread the 'Views' of some of my predecessors. One of the things I noticed is that occasionally a Chair would take the time to wax philosophical about a subject. So far, I have taken a different approach. I have tried to make my Views a chatty update on what's going on in the organization; to take the opportunity to thank people for jobs well done; and to ask for volunteers for one thing or another.

This month I'm going to deviate from my well-trod path, and attempt to emulate my predecessors and talk about something about which I have strong feelings.

On the value of mentoring

I recently hit another annual milestone in my career, passing my 36th year as a professional. As I am wont to do sometimes, I spent the anniversary of the start of my first real job thinking about the people with whom I worked all those years ago, and invariably I take some time to remember my mentor, Don Perkins.

Don was the lead software analyst on my first projects. Among many other things, Don guided me through my first really large scale, reusable software project; gave me my first opportunity to work with software that interfaces with custom data-acquisition hardware (which to this day remains my favorite type of work); sent me out on my first foray into solo field work; and gave my first opportunity to schedule and run a project [1].

One of my more vivid memories of our time working together occurred late in the afternoon one day. We were working on a system that used a different operating system than the one to which I was accustomed, and I was having trouble linking my program. I asked Don for help. He gave me a quick answer, and when I asked for details, he said to "just trust" him. I said that I did trust him, but I didn't completely understand the answer and if I just trusted him I'd never learn. Don took a deep breath, nodded his head, and proceeded to fully explain the answer he had given. We were both late leaving the office that night.

I can't recall Don ever hoarding information or knowledge, unlike some of the people with whom I've worked over the years. (There are few things I hate more than being told I don't need to know the answer to a question I have asked.) I've tried to emulate his example by freely, and sometimes expansively, providing help and transferring knowledge when asked.

Don also provided me with career advice from time to time. I was young and not finished with college when I started working for him; his perspectives on work and professionalism were invaluable.

I worked for Don for more than five years before moving on to a different position with the company. We continued to work together intermittently until I left the company to move to New Mexico, and he moved to California to support the large project on which he was working. I haven't seen him for more than 20 years, although we have exchanged emails from time to time.

Thanks, Don. If you didn't already, I hope you now realize how much of an impact your mentoring and friendship had on me in my early years.

One reason I've been thinking about mentoring is ACCU's emphasis on supporting organizations like Code Club [2] and Hour of Code [3]. Mentoring can take many forms; it is not limited to the type of mentoring I received. I encourage our membership to be a good mentor: to a child, a teen, a college student, a co-worker. The rewards are worth the effort.

History of ACCU

Matt Jones has been spearheading an effort to reconstruct the history of ACCU, concentrating on past committee members and honorary members. If you have been a committee member, or are an honorary member, please contact Matt with details of your service to ACCU (accumembership@accu.org). Dates of service are particularly helpful.

Call for volunteers

As this is being written, the web editor position has been vacant for more than a month. We are actively seeking one or more people to take on the responsibilities of the position. Until someone steps into the role, web site updates will be slower and less robust than they were while Martin Moene was web editor. Martin has indicated that the job takes about six hours a month, concentrated after the magazine is released. Please contact me if you are interested (chair@accu.org).

Notes and references

[1] 'A Scheduling Technique for Small Software Projects and Teams', *Overload* 123, October 2014, Page 123.
Don was the lead analyst on both of the projects mentioned in this article, and is listed in the acknowledgements.

[2] Code Club: www.codeclub.org.uk

[3] Hour of Code: https://hourofcode.com

JOIN THE ACCU!

You've read the magazine, now join the association dedicated to improving your coding skills.

The ACCU is a worldwide non-profit organisation run by programmers for programmers.

With full ACCU membership you get:

- 6 copies of *C Vu* a year
- 6 copies of *Overload* a year
- The ACCU handbook
- Reduced rates at our acclaimed annual developers' conference
- Access to back issues of ACCU periodicals via our web site
- Access to the *mentored developers projects*: a chance for developers at all levels to improve their skills
- Mailing lists ranging from general developer discussion, through programming language use, to job posting information
- The chance to participate: write articles, comment on what you read, ask questions, and learn from your peers.

Basic membership entitles you to the above benefits, but without Overload.

Corporate members receive five copies of each journal, and reduced conference rates for all employees.

How to join

You can join the ACCU using our online registration form. Go to **www.accu.org** and follow the instructions there.

Also available

You can now purchase exclusive ACCU T-shirts and polo shirts. See the web site for details.

Design: Pete Goodliffe

PERSONAL MEMBERSHIP
CORPORATE MEMBERSHIP
STUDENT MEMBERSHIP

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

GET MOORE
intel Software
intel
PARALLEL STUDIO XE
£634.99
TOOLS THAT EXTEND MOORE'S LAW
CREATE FASTER CODE—FASTER
Take your results to the next level with screaming-fast code.

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