the magazine of the accu

www.accu.org

{cvu}

Volume 25 • Issue 3 • July 2013 • £3

{cvu}

Volume 25 Issue 3
July 2013
ISSN 1354-3164
www.accu.org

Features Editor
Steve Love
cvu@accu.org

Regulars Editor
Jez Higgins
jez@jezuk.co.uk

Contributors
Frances Buontempo, Pete Goodliffe, Becky Grenier, Chris Oldwood, Roger Orr, Mark Radford, Vsevolod Vlaskine, Anthony Williams

ACCU Chair
Alan Griffiths
chair@accu.org

ACCU Secretary
Giovanni Asproni
secretary@accu.org

ACCU Membership
Mick Brooks
accumembership@accu.org

ACCU Treasurer
R G Pauer
treasurer@accu.org

Advertising
Seb Rose
ads@accu.org

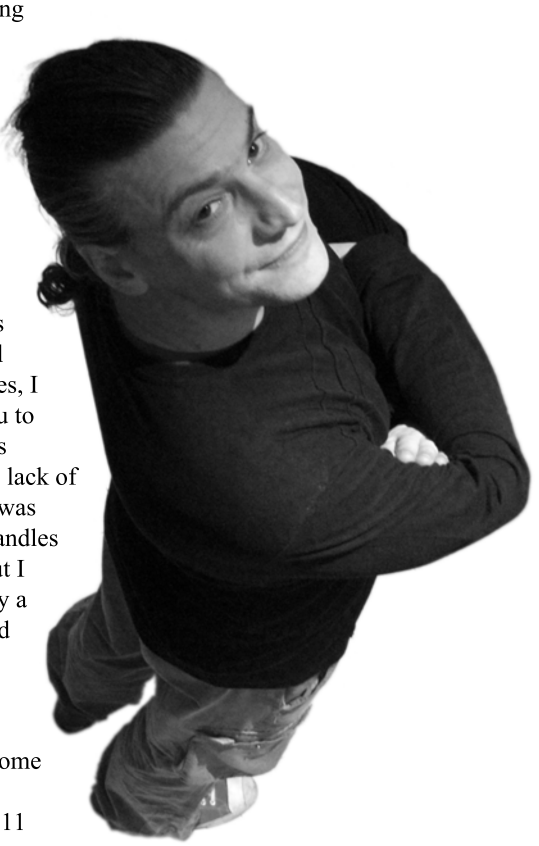
Cover Art
Pete Goodliffe

Print and Distribution
Parchment (Oxford) Ltd

Design
Pete Goodliffe

accu

A beginner again



I recently decided that it had been far too long since I'd written C++, at least more than a few lines to explore the syntax or behaviour of some feature or other. At the conference in particular, I tried hard to listen and comprehend knowledgeable people discussing features of C++11, but I'm afraid most of it went in one ear and out the other, for the simple reason that I don't really *think* in C++ any more.

I've spent most of the last decade writing lots of C# code, with just a couple of professional interludes requiring C++. In one of those cases, I was restricted to pre-1998 C++ (I'll leave you to guess which compiler for MS Windows I was required to use). In fact, for that problem, the lack of 'modern' features didn't really hinder me; it was really a pure OO design, and that compiler handles pure ABCs and virtual functions just fine. But I digress. I've become a C# programmer. I play a little with Python, a tiny bit of JavaScript, and can follow VBA code, but C# has been my main language for almost a dozen years.

So it was time to see if I could re-learn C++. Bjarne Stroustrup, on his home page and in some articles (available from http://www.stroustrup.com/) describes the 2011 revision of C++ as feeling like a whole new language, and I understand why that is. However, sitting down to write some real C++ code felt a little like getting back into an old chair; it took a little while to remember where the comfy parts are, and some of the worn bits with holes in, and the broken spring that sticks in your backside if you sit on the wrong bit. I won't say the memories came flooding back – more dribbling, to be honest – but I'm finding the exercise in recalling what I once knew, combined with learning new features a very satisfying experience.

And yes, all those years writing C# code now informs how I design and write C++ – a process that once went very much the other way.

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 25.4:	1st August 2013
C Vu 25.5:	1st October 2013
Overload 117:	1st September 2013
Overload 118:	1st November 2013

WRITE FOR C VU

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

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

ADVERTISE WITH US

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

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

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

COPYRIGHTS AND TRADE MARKS

On Software Design, Space, and Visuality

Vsevolod Vlaskine examines the motivations behind design visualisation.

Being a very visual person, I am not trying to say in this article that there is no merit in drawing diagrams as a part of software design and documentation. However, what I find really weird is the relatively common assumption that drawing pictures is universally good for software design, to the extent that a large part of the industry makes the graphic representation the centrepiece of the software design with the whole cottage industry around it.

As an example, for many years, UML has been promoted as the Unified Modeling Language, suggesting that it has all what you could possibly need for software design.

Perhaps it happens partly because software documentation without pictures is a notorious remedy for insomnia. Also, upper management and marketing love charts and drawings, especially with the drama of colour.

More importantly, the technical blueprint as a hallmark of the industrial era is expected to perform just as well in information technology.

The problem is that the picture essentially is a spacial representation. It portrays entities in space and therefore imposes a very particular view of the world (and ways of modelling it) as well as limits of what can be expressed in such a graphic language.

It starts with the suggestion to split the system into its major components, just like a car that consists of the frame, engine, wheels, etc. Then each component can be decomposed into smaller modules, etc. Technical drawings historically fit this purpose very well.

Then, the types of diagrams start to proliferate to assign visual symbols to various kinds of semantic relationships. The picture suggests the design where there are entities and relationships. Entities are represented by squares, matchstick men, time axes, etc. They roughly refer to a ‘place’ in the visual space. The relationships between the entities roughly correspond to ‘distances’ and ‘directions’ between ‘places’. (There is already a problem with such a break-down. The inventor of Pattern Languages, Christopher Alexander, demonstrated [1] that top-down analysis can represent complex heterogeneous systems only very poorly.)

Many highly efficient concepts are hard – but more importantly unnatural – to express in the world of entities and their relationships. Various transient objects, like scoped locks and transactions, may be good examples, because from a pragmatic point of view we would not think about them as objects, but rather qualities, interactions, semantic transformations and flows. Of course, we probably could invent diagrams to depict them, but why do it, if they find an essentially better expression in human or programming language?

In Jack W. Reeves’ words [2], believing that there can be a suitable visual representation for any aspect of software modelling is like being convinced that “‘different language’ really just means a different dialect of English”. To open a can of worms here: the native language of programming is the programming language. My experience has convinced me that the language of design is C, or C++, or python, etc, as opposed to UML. But going there would divert us too much from the critique of the visual metaphor as the purpose of this article.

UML comes from the early object-oriented paradigm in which the world is represented as a collection of types, instances and relationships. The limits of such an approach have been demonstrated by the semantic shift in modern languages. Take a look at the seminal C++ books by Meyers, Alexandrescu, Sutter, and others: there are not many pictures in them. Ask yourself why or try to draw diagrams that would adequately represent those concepts of the modern C++.

I wonder whether choosing UML as the preferred language for Software Patterns has made the books on them less read and patterns more often misunderstood, being considered not as a semiotic system, but as a list of illustrated recipies. Even Christopher Alexander’s book *A Pattern Language* [3], a dictionary of architectural patterns (what can be more visual?), does not contain too many illustrations.

When the translation between ideas and language gets funneled through a unified graphic representation, things get lost. Contrary to the conventional: “a picture is worth a thousand words”, a word can awaken a thousand pictures. While the picture merely explains, the word creates and transforms.

The visual in the Western tradition has been a vehicle of power and control. The tension the visual representation brings in is the one between visibility and transparency. They do not presume each other. Visibility is about total exposure, surveillance, fixed schematics for the purposes of accountability, unified control, the panopticon. Transparency is rather about not obstructing the flows. Flows mean not only efficient operation, but also the ability to morph and change.

In a strange twist, ‘visible’ and ‘transparent’ often become synonyms in process management, while in the vernacular ‘transparent’ means exactly the opposite: ‘invisible’. When visibility gets mistaken for transparency in a software team, design and documentation are required upfront, get divorced from the code, and very quickly lose their consistency with the functionality. With the latter drifting away from the document, ‘visible’ returns to its original connotation of ‘non-transparent’, ‘opaque’: the pictures and documentation do not reflect what the system does. They block the view.

One could say an organizational process is transparent, if nothing urges you to know its details, nothing makes you suspicious. Then, the next desired quality of a transparent process would be: as long as it flows smoothly, it is invisible, unnecessary to see. To achieve it, we need to get rid of the pictures as much as possible, remove them as a source of noise and instead focus on a good guarantee that only if something goes wrong, then the wrong part will become visible.

For example, test-driven development could be an example: we express the proper functioning and semantics of the system not through diagrams, but through test suits so that while they pass, we do not need to worry about them.

Similarly, in Scrum, as long as the burndown chart moves down smoothly, there is not much to talk about besides the brief scrum updates, but once the team observes a growing hump on it, it clearly indicates which part of the sprint has become a blockage. The low organizational overhead in Scrum not only saves time, but reduces noise in the system.

Approaching it from the other end: whatever needs to be visible or visualized should be suspicious and should not be taken on board without

VSEVOLOD VLASKINE

Vsevolod Vlaskine has over 15 years of programming experience. Currently, he leads a software team at the Australian Centre for Field Robotics, University of Sydney. He can be contacted at vsevolod.vlaskine@gmail.com

Passionate About Programming or Passionate About Life?

Chris Oldwood takes up the baton in the Passionate debate.

Life moves pretty fast. If you don't stop and look around once in a while, you could miss it.
~ Ferris Bueller.

There is a recurring topic that crops up at the ACCU Conference during the lightning talks and this year was no exception. In the first round of lightning talks Björn Fahller asked the question 'Why Are (Only) We Here?' And in the second set, Mike Long continued the trend with a talk entitled 'Passionate vs. Professional'.

Interestingly the use of the term 'passionate' was itself questioned by Seb Rose in his own 12" edition of the lightning talks with 'Are You Passionate?' For those who weren't there I suggest you look the word up in the dictionary and draw your own conclusions. For what it's worth I agree with Seb's sentiment, but I'm still going to stick to what I believe we mean by the word as it adds fuel to this particular fire…

So, let me ignite it now by suggesting that not everyone who works as a computer programmer does it because they are passionate about it. Yes, some of us cut our teeth on a home computer and are still bemused how we ended up getting paid to do what was essentially our hobby before entering employment. But that only applies to some of us. That's right; lots of people actually do a job for reasons other than the love of it.

each of us has a right to devote as much or as little time … to our career progression as we see fit

Many of the people I have worked with in the past do enjoy what they do. To go back to Seb's point about the word 'passionate', let me suggest (after clicking Shift+F7 to bring up the Thesaurus) the slightly watered down 'enthusiastic' instead. Programming is a career that they have chosen because they are genuinely interested in the subject. But I have also worked with others for whom programming was never really the end goal – they do it because it pays reasonably well (very well in certain industries) and perhaps they're better at it than other careers they originally had in mind. They are not, and never expect to be, The Best of the Best, but does every programmer have to aspire to be that?

When I went to university back in the late '80s I studied Electronic Systems Engineering. My choice was based on my dream of working in the Audio industry because I loved music and I loved my hi-fi. It turned out that I sucked really badly at analogue electronics! However I had slightly more success in the digital realm and eventually discovered that writing software was an actual profession, and one that I might be a little better (and therefore more successful) at.

I am one of the lucky ones who managed to make a course correction early in life and end up doing what I had already been doing for the 7 years prior to university, but hopefully in a somewhat more 'professional' manner. Of course not everyone is fortunate enough to have their path laid out clearly before them and so instead they fall into a job that pays the bills and then see where it goes. And so, if the mortgage is covered and it's not detestable, then why shouldn't that be enough for some?

There's that word 'professional' this time. What does that mean exactly? I can tell you I believe it doesn't include committing a load of changes that haven't even been compiled just before going off on a week's holiday. But what about some of the practices that many of us feel are beneficial to a sustainable system, say, test-first versus test-later or continuous integration? What's the penance

CHRIS OLDWOOD

Chris is a freelance developer who started out as a bedroom coder in the 80's writing assembler on 8-bit micros. These days it's C++ and C# on Windows in big plush corporate offices. He is also the commentator for the Godmanchester Gala Day Duck Race and can be contacted via gort@cix.co.uk or @chrisoldwood

Software Design, Space, and Visuality (continued)

a good reason. Things are visible because they are opaque, which means that they represent a solid structure capable of blocking flows.

For example, stakeholders tend to perceive Gantt charts as the true image of resource allocation. However, any non-trivial project has too much uncertainty and simply does not flow exactly according to the plan. Gantt charts misrepresent the project and introduce milestones or deliveries at the wrong places.

Over-engineered applications or over-structured teams require visualizations exactly because they are opaque, blocking flows and lines of sight too easily. (One almost could consider visual design as a kind of negative space, a system of points of blockage and failure – and effectively that is what test-driven development is a remedy for.)

Software artifacts and the design process are functional and flow-oriented. Therefore, non-obstructing flow (in any sense: code flow, data flow, flow of design activities, human communication flow) is more essential than structural visibility. Also, they have close affinity to change and time, not only its linear axis, but also its qualitative, transformational structure, which is not necessary linear. Take 'software life-cycle' as a colloquial example of cyclic time structures in software design; or 'time-to-market' expressing not just linear time, but a finite strongly structured segment of it. Spatial representation of qualitative aspects of time is doubtful, while languages (natural or artificial) have been developing exactly to reflect and express such temporal qualities.

Of course, everything changes, when pictures and diagrams are used as transient, disposable, molecular, optional elements of collaboration, much like words of the spoken, written, or programming language. ■

References

[1] Christopher Alexander, A City Is Not a Tree. Architectural Forum, Vol 122. 1965 http://www.rudi.net/pages/8755

[2] Jack W. Reeves, 'What Is Software Design: 13 Years Later' *Developer.* Magazine*. 23 February 2005. http://www.developerdotstar.com/mag/articles/reeves_13yearslater.html

[3] Christopher Alexander et al, *A Pattern Language: Towns, Buildings, Construction*. 1977

for not adopting these and continuing to work in an 'old fashioned' way? If the goal is to deliver working software then surely how that's achieved is of secondary importance to if it's achieved?

I once worked with a chap at a big corporation who was purely in 'the programming profession' to fund his non-working life. This was quite an eye opener for me as I couldn't fault his rationale even though the inefficiencies of his development style used to annoy the hell out of me. I tried (largely in vain) to evangelise about the virtues of certain modern development practices but ultimately they fell on deaf ears. His performance wasn't brilliant, but via perspiration he would end up delivering something that works. Is it 'unprofessional' to get the job done in ways that perhaps not everyone agrees on? As his peer I had to leave him to it; just so long as what he did had no direct affect on my ability to deliver.

You can of course argue that poor design, lack of tests, blah, blah, blah all leads to an increased delivery time for subsequent changes and so it will eventually affect my ability to make changes. But I do not feel that is of my concern – that is a management issue to me. Like it or not software development is rife with politics and the 'performance' of an individual is just one facet that determines the make-up of a team. Much to the chagrin of one manager I worked with, you can't just fire someone because you've found a better replacement. Anyone who has ever worked in a large corporation will wince when I mention the term 'head count' – a pejorative used to describe why you sometimes have to settle for the 10th best interview candidate instead of holding out for the 1st or 2nd.

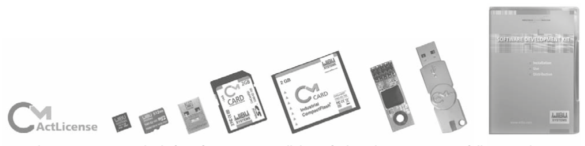
In (my) ideal world there would be a causal relationship between 'coolness' of project, quality of team members, and rate of pay; with them all increasing together. Sadly it's more likely that there is an inverse square law in effect somewhere as there are industries that seem to rely on youth and adoration to compensate for a lack of remuneration. Conversely you'd expect Finance, with its deeper pockets, to only hire the best-of-the-best; but I can assure you that hypothesis doesn't hold water either.

Coming back to the original question then, I wonder if there is an expectation that ACCU conference attendees (or SPA, QCon, etc.) are considered by some as 'the norm', whereas those who choose explicitly not to go are somehow 'exceptional' for not wanting to better themselves? There is a third group of people who are even unaware of such events, but I'm guessing the most curiosity comes from those who choose to ignore our evangelism rather than embrace it. I don't believe anyone genuinely goes out of their way to be a Luddite, although people can become disgruntled when they believe they have been treated unfairly. Instead I've come to see my fellow programmers as people with different abilities, and more importantly, with different priorities. They all have a desire to learn and improve; it's just that they've chosen a rate that is much lower than we have chosen for ourselves. Is it 'unprofessional' to be a slow learner?

I'm sure I didn't fool anyone with the purposefully controversial title that suggests it's impossible to enjoy a career in programming and at the same time enjoy a fruitful life. In essence it's a thinly veiled attempt to re-cast the age old debate of 'live to work' or 'work to live' into a programming context. Clearly each of us has a right to devote as much or as little time, outside our normal working day, to our career progression as we see fit.

What I'm going to suggest, and I'm sure I'll be labelled a heretic for suggesting it, is that the next time you are presented with the question of why you are there, is to first ask yourself why you're not snowboarding on the Alps, lazing on the beach soaking up the rays or curled up on the sofa reading a trashy novel? ■

CodeMeter®
Licensing made Easy



CodeMeter comes in multiple form factors to meet all diversified needs Start testing its full potential now:
Order CodeMeter SDK at www.wibu.com/uk/sdk

- Best in class security features
- Global or individual license control
- Hardware and software licenses
- Multi-platform solution

Mac Universal

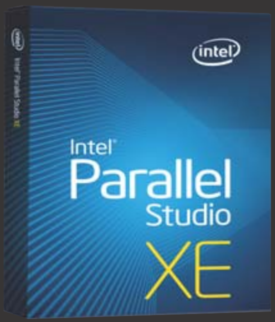
MEDIA
ACCESS
PERFECTION IN SOFTWARE PROTECTION
DOCUMENT

WIBU SYSTEMS

WIBU-SYSTEMS LTD | The Mansion Bletchley Park | MK3 6DS Bletchley, Milton Keynes, United Kingdom
+44 20 314 747 27 | sales@wibu.co.uk | www.wibu.com

intel

Introducing Intel Parallel Studio XE 2013



Develop top application performance while minimising development, tuning and testing time and effort.

Intel® Parallel Studio XE combines:

- Industry-leading compilers
- Performance and parallel libraries
- Error checking tools
- Performance profiling tools

qbs SOFTWARE *We make it happen*

www.qbssoftware.com/ipsxe2013
08456 580 580
sales@qbssoftware.com

All the World's a Stage...

Anthony Williams shows how Actors simplify multithreaded code in C++11.

Handling shared mutable state is probably the single hardest part of writing multithreaded code. There are lots of ways to address this problem; one of the common ones is the actors metaphor. Going back to Hoare's *Communicating Sequential Processes* [1], the idea is simple – you build your program out of a set of actors that send each other messages. Each actor runs normal sequential code, occasionally pausing to receive incoming messages from other actors. This means that you can analyse the behaviour of each actor independently; you only need to consider which messages might be received at each receive point. You could treat each actor as a state machine, with the messages triggering state transitions.

This is how Erlang processes work: each process is an actor, which runs independently from the other processes, except that they can send messages to each other. **Just::thread Pro: Actors Edition** [2] adds library facilities to support this to C++. In the rest of this article I will describe how to write programs that take advantage of it. Though the details will differ, the approach can be used with other libraries that provide similar facilities, or with the actor support in other languages.

Simple actors

Actors are embodied in the `jss::actor` class. You pass in a function or other callable object (such as a lambda function) to the constructor, and this function is then run on a background thread. This is exactly the same as for `std::thread`, except that the destructor waits for the actor thread to finish, rather than calling `std::terminate`.

```
void simple_function(){
  std::cout<<"simple actor\n";
}

int main(){
  jss::actor actor1(simple_function);
  jss::actor actor2([]{
    std::cout<<"lambda actor\n";
  });
}
```

The waiting destructor is nice, but it's really a side issue – the main benefit of actors is the ability to communicate using messages rather than shared state.

Sending and receiving messages

To send a message to an actor you just call the `send()` member function on the actor object, passing in whatever message you wish to send. `send()` is a function template, so you can send any type of message – there are no special requirements on the message type. You can also use the stream-insertion operator to send a message, which allows easy chaining e.g.

```
actor1.send(42);
actor2.send(MyMessage("some data"));
actor2<<Message1()<<Message2();
```

ANTHONY WILLIAMS

Anthony is the author of C++ Concurrency in Action. He has worked in a myriad of languages over the years, and enjoys the challenge of solving new problems. He can be contacted at anthony@justsoftwaresolutions.co.uk

Sending a message to an actor just adds it to the actor's message queue. If the actor never checks the message queue then the message does nothing. To check the message queue, the actor function needs to call the `receive()` static member function of `jss::actor`. This is a static member function so that it always has access to the running actor, anywhere in the code – if it were a non-static member function then you would need to ensure that the appropriate object was passed around, which would complicate interfaces, and open up the possibility of the wrong object being passed around, and lifetime management issues.

The call to `jss::actor::receive()` will then block the actor's thread until a message that it can handle has been received. By default, the only message type that can be handled is `jss::stop_actor`. If a message of this type is sent to an actor then the `receive()` function will throw a `jss::stop_actor` exception. Uncaught, this exception will stop the actor running. In the following example, the only output will be "Actor running", since the actor will block at the `receive()` call until the stop message is sent, and when the message arrives, `receive()` will throw.

```
void stoppable_actor(){
  std::cout<<"Actor running"<<std::endl;
  jss::actor::receive();
  std::cout<<"This line is never run"<<std::endl;
}

int main(){
  jss::actor a1(stoppable_actor);
  std::this_thread::sleep_for
    (std::chrono::seconds(1));
  a1.send(jss::stop_actor());
}
```

Sending a "stop" message is common-enough that there's a special member function for that too: `stop()`. `a1.stop()` is thus equivalent to `a1.send(jss::stop_actor())`.

Handling a message of another type requires that you tell the `receive()` call what types of message you can handle, which is done by chaining one or more calls to the `match()` function template. You must specify the type of the message to handle, and then provide a function to call if the message is received. Any messages other than `jss::stop_actor` not specified in a `match()` call will be removed from the queue, but otherwise ignored. In the following example, only messages of type `int` and `std::string` are accepted; the output is thus:

```
Waiting
42
Waiting
Hello
Waiting
Done
```

The code is in Listing 1.

It is important to note that the `receive()` call will block until it receives one of the messages you have told it to handle, or a `jss::stop_actor` message, and unexpected messages will be removed from the queue and discarded. This means the actors don't accumulate a backlog of messages they haven't yet said they can handle, and you don't have to worry about out-of-order messages messing up a `receive()` call.

These simple examples have just had `main()` sending messages to the actors. For a true actor-based system we need them to be able to send

Listing 1

```
void simple_receiver(){
  while(true){
    std::cout<<"Waiting"<<std::endl;
    jss::actor::receive()
      .match<int>([](int i){
         std::cout<<i<<std::endl;})
      .match<std::string>([]
         (std::string const&s){
           std::cout<<s<<std::endl;});
  }
}

int main(){
  {
    jss::actor a(simple_receiver);
    a.send(true);
    a.send(42);
    a.send(std::string("Hello"));
    a.send(3.141);
    a.send(jss::stop_actor());
  } // wait for actor to finish
  std::cout<<"Done"<<std::endl;
}
```

messages to each other, and reply to messages. Let's take a look at how we can do that.

Referencing one actor from another

Suppose we want to write a simple time service actor, that sends the current time back to any other actor that asks it for the time. At first thought it looks rather simple: write a simple loop that handles a 'time request' message, gets the time, and sends a response. It won't be that much different from our **simple_receiver()** function in Listing 1:

```
struct time_request{};
void time_server(){
  while(true){
    jss::actor::receive()
      .match<time_request>([](time_request r){
        auto now=std::chrono::system_clock::now();
        ????.send(now);
      });
  }
}
```

Listing 3

```
void time_server(){
  while(true){
    jss::actor::receive()
      .match<time_request>([](time_request r){
        auto
          now=std::chrono::system_clock::now();
        try{
          r.sender<<now;
        } catch(jss::no_actor&){}
      });
  }
}
```

The problem is, we don't know which actor to send the response to – the whole point of this time server is that it will respond to a message from *any* other actor. The solution is to pass the sender as part of the message. We could just pass a pointer or reference to the **jss::actor** instance, but that requires that the actor knows the location of its own controlling object, which makes it more complicated – none of the examples we've had so far could know that, since the controlling object is a local variable declared in a separate function. What is needed instead is a simple means of identifying an actor, which the actor code can query – an actor reference. The type of an actor reference is **jss::actor_ref,** which is implicitly constructible from a **jss::actor**. An actor can also obtain a reference to itself by calling **jss::actor::self()**. **jss::actor_ref** has a **send()** member function and stream insertion operator for sending messages, just like **jss::actor**. So, we can put the sender of our **time_request** message in the message itself as a **jss::actor_ref** data member, and use that when sending the response (see Listing 2).

If you use **jss::actor_ref** then you have to be prepared for the case that the referenced actor might have stopped executing by the time you send the message. In this case, any attempts to send a message through the **jss::actor_ref** instance will throw an exception of type **jss::no_actor**. To be robust, our time server really ought to handle that too – if an unhandled exception of any type other than **jss::stop_actor** escapes the actor function then the library will call

Listing 2

```
struct time_request{
  jss::actor_ref sender;
};
void time_server(){
  while(true){
    jss::actor::receive()
      .match<time_request>([](time_request r){
        auto now=
          std::chrono::system_clock::now();
        r.sender<<now;
      });
  }
}
void query(jss::actor_ref server){
  server<<time_request{jss::actor::self()};
  jss::actor::receive()
    .match<std::chrono::system_clock::
        time_point>(
      [](std::chrono::system_clock::time_point){
        std::cout<<"time received"<<std::endl;
    });
}
```

Listing 4

```
struct pingpong{
  jss::actor_ref sender;
};
void pingpong_player(std::string message){
  while(true){
    try{
      jss::actor::receive()
        .match<pingpong>([&](pingpong msg){
          std::cout<<message<<std::endl;
          std::this_thread::sleep_for
            (std::chrono::milliseconds(50));
          msg.sender<<pingpong{
            jss::actor::self()};
        });
    }
    catch(jss::no_actor&){
      std::cout<<"Partner quit"<<std::endl;
      break;
    }
  }
}
int main(){
  jss::actor ping(pingpong_player,"ping");
  jss::actor pong(pingpong_player,"pong");
  ping<<pingpong{pong};
  std::this_thread::sleep_for
    (std::chrono::seconds(1));
  ping.stop();
  pong.stop();
}
```

std::terminate. We should therefore wrap the attempt to send the message in a try-catch block (Listing 3).

We can now set up a pair of actors that play ping-pong (Listing 4).

This will give output along the lines of the following:

```
ping
pong
ping
pong
ping
pong
ping
pong
ping
pong
ping
pong
ping
pong
ping
pong
ping
pong
ping
pong
Partner quit
```

The sleep in the player's message handler is to slow everything down – if you take it out then messages will go back and forth as fast as the system can handle, and you'll get thousands of lines of output. However, even at full speed the pings and pongs will be interleaved, because sending a message synchronizes with the **receive()** call that receives it.

That's essentially all there is to it – the rest is just application design. As an example of how it can all be put together, let's look at an implementation of the classic sleeping barber problem.

The Lazy Barber

For those who haven't met it before, the problem goes like this: Mr Todd runs a barber shop, but he's very lazy. If there are no customers in the shop then he likes to go to sleep. When a customer comes in they have to wake him up if he's asleep, take a seat if there is one, or come back later if there are no free seats. When Mr Todd has cut someone's hair, he must move on to the next customer if there is one, otherwise he can go back to sleep.

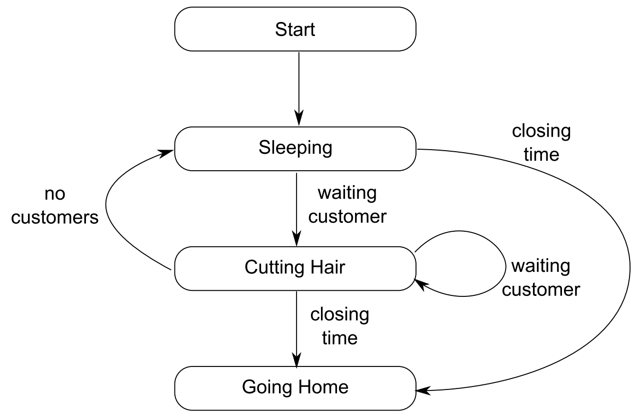
Let's start with the barber. He sleeps in his chair until a customer comes in, then wakes up and cuts the customer's hair. When he's done, if there is a waiting customer he cuts that customer's hair. If there are no customers, he goes back to sleep, and finally at closing time he goes home. This is shown as a state machine in Figure 1.

This translates into code as shown in Listing 5.The **wait** loops for 'sleeping' and 'cutting hair' have been combined, since almost the same set of messages is being handled in each case – the only difference is that the 'cutting hair' state also has the option of 'no customers', which cannot be received in the 'sleeping' state, and would be a no-op if it was. This allows the action associated with the 'cutting hair' state to be entirely handled in the lambda associated with the **customer_waiting** message; splitting the wait loops would require that the code was extracted out to a separate function, which would make it harder to keep count of the haircuts. Of course, if you don't have a compiler with lambda support then you'll need to do that anyway. The logger is a global actor that receives **std::string**s as messages and writes the to **std::cout**. This avoids any synchronization issues with multiple threads trying to write out at once, but it does mean that you have to pre-format the strings, such as when logging the number of haircuts done in the day. The code for this is shown in Listing 6.

Let's look at things from the other side: the customer. The customer goes to town, and does some shopping. Each customer periodically goes into the barber shop to try and get a hair cut. If they manage, or the shop is closed, then they go home, otherwise they do some more shopping and go back to the barber shop later. This is shown in the state machine in Figure 2.

This translates into the code in Listing 7. Note that the customer interacts with a 'shop' actor that I haven't mentioned yet. It is often convenient to

Figure 1



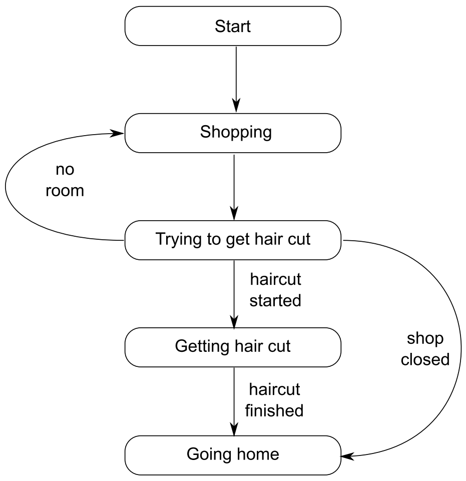
Listing 5

```
void barber_func()
{
  bool go_home=false;
  unsigned haircuts=0;
  while(!go_home)
  {
    logger<<std::string("barber is sleeping");
    bool can_sleep=false;
    do
    {
      jss::actor::receive()
        .match<customer_waiting>(
          [&](customer_waiting c){
            logger<<std::string
               ("barber is cutting hair");
            c.customer<<start_haircut();
            std::this_thread::sleep_for
              (std::chrono::milliseconds
                (1000*(1+(rand()%5))));
            c.customer<<done_haircut();
            ++haircuts;
          })
        .match<no_customers>(
          [&](no_customers){
            can_sleep=true;
          })
        .match<closing_time>(
          [&](closing_time){
            go_home=true;
          });
    }
    while(!can_sleep && !go_home);
  }
  std::ostringstream os;
  os<<"barber is going home. He did
    "<<haircuts<<" haircuts today";
  logger<<os.str();
}
```

Listing 6

```
void logger_func(){
  for(;;){
    jss::actor::receive()
      .match<std::string>([](std::string s){
        std::cout<<s<<std::endl;
      });
  }
}
jss::actor logger(logger_func);
```

Figure 2



have an actor that represents shared state, since this allows access to the shared state from other actors to be serialized without needing an explicit mutex. In this case, the shop holds the number of waiting customers, which must be shared with any customers that come in, so they know whether there is a free chair or not. Rather than have the barber have to deal with messages from new customers while he is cutting hair, the shop acts as an intermediary. The customer also has to handle the case that the shop has already closed, so the **shop** reference might refer to an actor that has finished executing, and thus get a **jss::no_actor** exception when trying to send messages.

Listing 7

```
enum class haircut_status{
  had_haircut,no_room,shop_closed
};
haircut_status try_and_get_hair_cut
    (unsigned customer,jss::actor_ref shop){
  std::ostringstream os;
  os<<"customer "<<customer<<
    " goes into barber shop";
  logger<<os.str();
  try{
    shop<<customer_enters{
      jss::actor::self()};
  }
  catch(jss::no_actor){
    os.str("");
    os<<"customer "<<customer<<
      " finds barber shop is closed";
    logger<<os.str();
    return haircut_status::shop_closed;
  }
  haircut_status status=haircut_status::no_room;
  jss::actor::receive()
    .match<start_haircut>(
      [&](start_haircut)
      {
        os.str("");
        os<<"customer "<<customer<<
          " is having a haircut";
        logger<<os.str();
        jss::actor::receive()
          .match<done_haircut>(
            [&](done_haircut)
            {
              os.str("");
              os<<"customer "<<customer<<
                " is done having a haircut";
              logger<<os.str();
            }
            );
        status=haircut_status::had_haircut;
      }
      )
```

Listing 7 (cont'd)

```
    .match<no_room>(
      [&](no_room)
      {
        os.str("");
        os<<"customer "<<customer<<
          " leaves because there is no room";
        logger<<os.str();
        status=haircut_status::no_room;
      }
      )
    .match<shop_closed>(
      [&](shop_closed)
      {
        os.str("");
        os<<"customer "<<customer<<
          " finds barber shop is closed";
        logger<<os.str();
        status=haircut_status::shop_closed;
      }
      );
  os.str("");
  os<<"customer "<<customer<<
    " leaves barber shop";
  logger<<os.str();
  try{
    shop<<customer_leaves();
  }
  catch(jss::no_actor){
  }
  return status;
}
void customer_func(unsigned i,
                   jss::actor_ref shop)
{
  std::ostringstream os;
  os<<"customer "<<i<<" goes to town";
  logger<<os.str();
  haircut_status status;
  do{
    os.str("");
    os<<"customer "<<i<<" is shopping";
    logger<<os.str();
    std::this_thread::sleep_for
      (std::chrono::milliseconds
       (500*(rand()%20)));
  }
  while((status=try_and_get_hair_cut(i,shop))
    ==haircut_status::no_room);
  os.str("");
  os<<"customer "<<i<<" is going home";
  logger<<os.str();
}
```

The message handlers for the shop are short, and just send out further messages to the barber or the customer, which is ideal for a simple state-manager – you don't want other actors waiting to perform simple state checks because the state manager is performing a lengthy operation; this is why we separated the shop from the barber. The shop has 2 states: open, where new customers are accepted provided there are fewer than the remaining spaces, and closed, where new customers are turned away, and the shop is just waiting for the last customer to leave. If a customer comes in, and there is a free chair then a message is sent to the barber that there is a customer waiting; if there is no space then a message is sent back to the customer to say so. When it's closing time then we switch to the 'closing' state – in the code we exit the first **while** loop and enter the second. This is all shown in listing 8.

Listing 8

```
void shop_func(jss::actor_ref barber)
{
  bool closed=false;
  unsigned waiting_customers=0;
  unsigned const max_waiting_customers=3;

  std::ostringstream os;
  logger<<std::string("shop opens");

  while(!closed)
  {
    jss::actor::receive()
      .match<customer_enters>(
        [&](customer_enters c){
          ++waiting_customers;
          os.str("");
          os<<"shop has "<<waiting_customers<<"
            customers";
          logger<<os.str();
          if(waiting_customers<=
             max_waiting_customers){
            barber<<customer_waiting{c.customer};
          } else
            c.customer<<no_room();
        })
      .match<customer_leaves>(
        [&](customer_leaves){
          if(!--waiting_customers)
          {
            logger<<"last customer left shop";
            barber<<no_customers();
          } else{
            os.str("");
            os<<"shop has
             "<<waiting_customers<<" customers";
            logger<<os.str();
          }
        })
      .match<closing_time>(
        [&](closing_time c){
          logger<<std::string("shop closing");
          closed=true;
          barber<<c;
        });
  }

  while(waiting_customers){
    os.str("");
    os<<"shop has "<<waiting_customers<<"
       customers";
      logger<<os.str();
    jss::actor::receive()
      .match<customer_enters>(
        [&](customer_enters c){
          ++waiting_customers;
          logger<<"customer turned away
             because shop closed";
          c.customer<<shop_closed();
        })
      .match<customer_leaves>(
        [&](customer_leaves){
          if(!--waiting_customers)
          {
            logger<<"last customer left shop";
          }
        });
  }
  logger<<std::string("shop closed");
}
```

Listing 9

```
struct customer_waiting
{
  jss::actor_ref customer;
};
struct customer_enters
{
  jss::actor_ref customer;
};
struct customer_leaves
{};
struct start_haircut
{};
struct done_haircut
{};
struct closing_time
{};
struct no_room
{};
struct shop_closed
{};
struct no_customers
{};
```

The messages are shown in listing 9, and the **`main()`** function that drives it all is in listing 10.

Exit stage left

There are of course other ways of writing code to deal with any particular scenario, even if you stick to using actors. This article has shown some of the issues that you need to think about when using an actor-based approach, as well as demonstrating how it all fits together with the **Just::Thread Pro** actors library. Though the details will be different, the larger issues will be common to any implementation of the actor model. ■

References

[1] *Communicating Sequential Processes*, C.A.R Hoare, 1985. http://www.usingcsp.com/
[2] Just::Thread Pro: Actors Edition, http://www.stdthread.co.uk/pro/

Listing 10

```
int main()
{
  {
    jss::actor barber(barber_func);
    jss::actor barbershop
      (shop_func,jss::actor_ref(barber));
    unsigned const count=20;
    jss::actor customers[count];

    for(unsigned i=0;i<count;++i)
    {
      std::ostringstream os;
      os<<"Starting customer "<<i;
      logger<<os.str();
      customers[i]=jss::actor(customer_func,i,
        jss::actor_ref(barbershop));
    }

    std::this_thread::sleep_for
      (std::chrono::seconds(20));
    barbershop<<closing_time();
  }

  logger.stop();
}
```

How I Wrote My First Technical Presentation

Becky Grenier shares her preparations for giving a tech talk.

This was a letter to the DevChix (www.devchix.org) mailing list, a technical community for and by women in software development. My thanks to Becky Grenier for agreeing to share it with ACCU, where I hope it'll inspire people in our community, too.

"Just write a great description for your talk, send it in, and then you'll have no choice but to pull it together sometime before the conference." As I gave this advice to a friend, I saw that this could be the answer to my own public speaking aversion as well. If I waited until after I had put together a great presentation it was just never going to happen.

So, a few weeks later I sent an email to a local user group suggesting a topic I had recently learned quite a bit about and implemented for work, the Apache Solr Search Server. They were very receptive and we set a date about one month away, on which I would give the presentation.

The panic set in two weeks from the presentation date. I thought I should learn more about Apache Solr and tried to read the book I had, but it was more like desperate scanning since I didn't really have enough time for that anymore. Then, when I saw the date of my talk was a little over a week away I began on the slides. It was the first time I had used PowerPoint in over a decade. I had no idea how to start so I just forced myself to keep making slides until I had about 15, which I thought was a good start for a 30-minute presentation. This took quite a few hours and quite a few beers. They were terrible slides and I knew it, basically just lists of bullet-points. And my self-doubt kept walloping me over the head after each one saying, "this sucks, your topic sucks and so do you". Which is hard to get past when the work you are doing does actually suck. And so it was with great difficulty that I finished my first draft.

A couple of saintly friends of mine were willing to sit through my first practice. Not only were they terrible slides, but it was a boring topic as well and I was not a good presenter. They stopped me shortly after I had begun and said they were having trouble following. I hadn't defined several terms. I hadn't really said what Apache Solr was. I hadn't explained why someone might want to use it. It was hard for them to pay attention when they didn't know the relevance. It was just me talking about configuration files. Instead, they said, I should tell a story. (I thought, 'Once upon a time there was this software program...') They were right, and I had to start the whole thing over again, which I did that very night, with a few less beers this time.

Thus began my endless rounds of practising, and then tearing apart my slides. I practised in front of both technical and non-technical people. I ended up defining every technical term I used, even things I was pretty sure my audience would know, such as 'API'. As my sister told me, it takes 10 seconds to offer a quick definition and it is a kindness to anyone who might not know. Slowly my presentation took on a shape with an introduction, middle, and conclusion, and with all necessary pieces such as 'About Me' and 'Questions'. I allowed the rising panic I felt as the date approached to drive me to keep practising. I got pretty good at explaining my slides. I had even put in a few that made people laugh. I practised six times in all.

On the day of my presentation, I was nervous still (maybe the panic had just become habit by then) but felt more confident due to all the time I had put in. If I didn't do well, at least I had given it my best shot, but I was pretty sure I had beat that presentation into something decent. Finally it was time to speak, and it went pretty well. The audience laughed at the appropriate spots. A few times I felt like my knees were trembling but the feedback I got was that I didn't seem nervous (evidence that nobody can tell). At the last minute I had decided to demo an app made with that technology so they could see it in action, and that ended up not working, but I was able to skip over that pretty quickly. It was just an extra anyway, not fundamental to understanding my topic. I told my audience I was a new public speaker and asked for any feedback they might have (something I read in 'Lean In'), and everyone said I did great and there was nothing negative.

Then I was so relieved to be done and have this monkey of a presentation off my back. I was proud of myself for doing something that scared me so much, and I had created a pretty good talk that I could give again. It was surprising, not only to me but to those friends who had watched my initial efforts, how much improved my final presentation was – almost unbelievable. And I have volunteered to give it again already, to a different group in a few weeks, and I'm not that nervous about it anymore.

So the moral to this story is that all you have to do to become a public speaker is just find somewhere to start and keep going. ■

BECKY GRENIER

Rebecca Grenier is a Software Developer for *EatingWell Magazine* in Vermont, USA. She began programming at age 12, when she and her twin sister created infinite loops in GW BASIC to insult each other repeatedly. She can be reached at rebeccagrenier@gmail.com

grey matter®
software know how
CELEBRATING 30 YEARS!

Receive expert advice on

Visual Studio · intel Software Elite Reseller · embarcadero reseller partner

and more...

For all your software developer tools call Grey Matter on **01364 654100** or visit **www.greymatter.com**

Wrapper Scripts

Chris Oldwood automates his toolkit for an easier and more predictable life.

At the ACCU conference this year, Pete Goodliffe hosted a session titled 'Becoming a Better Programmer'. Part of it involved a number of people (that Pete had invited) spending a few minutes describing what they believe has helped make them a better programmer. One such person was the editor of this very journal – Steve Love – who picked Automation as his topic. If you read his editorial from a few issues back you'll know that, like Pete and Steve, I too prefer to simplify things when I find myself doing the same task over and over. The full subjects of both automation and scripting are huge in themselves, but there is a particular intersection that at first might seem almost trivial and yet can quickly grow into something more useful – wrapper scripts.

One tried and trusted solution to these 'problems' is to turn to the venerable Wiki and document lots of command lines as snippets

Simplifying existing tools

If you've ever worked with SQL Server from the command line you'll have come across the bundled SQLCMD.EXE tool. This, along with its forerunner OSQL.EXE, is the traditional tool for executing SQL statements (and script files) against a SQL Server instance. Like many mature command line tools, it's a bit of a Swiss-Army knife and has sprouted a myriad of options and switches to control how you feed SQL text into it and how the results and errors are handled after execution. For example the following command will fetch the current time on a local instance using the current user credentials:

```
C:\> SQLCMD -E -S .\SQLEXPRESS -d master-b -m 10
-Q "select getdate();"
```

The **-b** and **-m** switches are technically unnecessary when running interactively, but the moment you start running SQL batches from scripts you'll likely add them if you want anything out of the ordinary to cause execution to stop and SQLCMD to return an error code. Then there is the annoyance factor of just getting the command line slightly wrong. If you forget the **-E** you'll get a weird login failure, or if you use **-q** instead of **-Q** it won't terminate after executing the SQL. Case-sensitive command line tools do nothing to help ensure a calm, quiet working environment either.

One tried and trusted solution to these 'problems' is to turn to the venerable Wiki and document lots of command lines as snippets that you can just cut-and-paste directly into your shell. Anyone who has ever tried that from a Word document where Word has been 'smart' and converted the simple dashes to 'smart' dashes will know how fraught with danger this idea is. That's before you consider what happens when the wiki becomes unavailable (and I can guarantee a development server has an SLA measured in months) or you begin to appreciate the shear tediousness of what it is you're actually doing.

Another more personal alternative is to use your shell or some 3rd party tool to create little macros for your favourite commands. However, I feel this is a bit like writing your own tests – if it's good enough for you, then why not the rest of the team? After all, when a new team member joins they're probably going to have to go through the same process. So, you can give them a leg-up by storing a set of project-specific pre-canned scripts that help with the most common scenarios.

In a sense it's a bit like Partial Function Application because, whereas a tool like SQLCMD has to allow for the different modes of operation in general, your development environment will almost certainly be far more specific. This means you can exploit that knowledge by first cutting down on any superfluous arguments. The command below, for example, creates a new database on a local instance:

```
C:\> SQLCMD -E -S .\SQLEXPRESS -d master -Q
"create database MyTestDb;"
```

The only real variables in the command are the instance (**.\SQLEXPRESS**) and the database name (**MyTestDb**), the rest is boilerplate stuff. So, let's create a batch file that accepts just those two parameters, and more importantly has a memorable name (`CreateDatabase.cmd`):

```
@SQLCMD -E -S %1 -d master -Q "create database
%2;"
```

Now we can just use this simple script to create a test database in future:

```
C:\> CreateDatabase .\SQLEXPRESS MyTestDb
```

Error handling

Of course just like any other code we write we have to consider the handling of errors and so we should add a sprinkling of that too – first to check we have the required number of arguments, then to pass back any error returned by the actual tool:

```
@echo off
if /i "%1" == "" call :usage & exit /b 1
if /i "%2" == "" call :usage & exit /b 1

SQLCMD -E -S %1 -d master -Q "create database %2;"
if errorlevel 1 exit /b %errorlevel%

:usage
echo Usage: %~n0 ^<instance^> ^<database^>
goto :eof
```

If you have bothered to write a usage message, then you could also choose to add a couple of extra lines to provide a consistent and modern way to query it (although this somewhat starts to defeat the original purpose of the exercise – simplification!):

```
if /i "%1" == "-?"     call :usage & exit /b 0
if /i "%1" == "--help" call :usage & exit /b 0
```

Now, at this point I'm sure you're beginning to question whether the script is starting to get so complicated that you'll be spending more time writing it than what you (and hopefully your colleagues) will eventually save by using it. To decide that I suggest you consult the recent XKCD titled 'Is It Worth the Time?' [1].

CHRIS OLDWOOD

Chris is a freelance developer who started out as a bedroom coder in the 80's writing assembler on 8-bit micros. These days it's C++ and C# on Windows in big plush corporate offices. He is also the commentator for the Godmanchester Gala Day Duck Race and can be contacted via gort@cix.co.uk or @chrisoldwood

However it should also be fairly obvious that this is just boilerplate code that can be copied from a template. One such template (for batch files) can be found in my blog post ‘Windows Batch File Template’ [2]. Consequently I’ve found putting these kind of scripts together quite trivial and it also allows for some other common simple scenarios to be easily accommodated.

Personalisation

As a rule of thumb the point of these sorts of scripts is to allow a more unified development experience through the use of some common tools. Also the development environment should allow for the redundancy to be hidden or removed as we saw above. But there are times when each developer (and the build machine itself counts as another developer) needs to provide some personal configuration data.

Once again the database example fits the bill nicely. In many organisations you can install a database locally, but not everywhere allows this (you can thank an overreaction to the 2003 Slammer virus for that). Although not ideal, you may also have to deal with minor differences in development machine configuration such as where the instance is installed (a fresh install and an upgrade can be different). For example nearly every developer may have a local copy of SQL Server Express and be quite happy calling their unit test database ‘UnitTest’, in which case you might as well save them a bit more typing and just default to those values:

```
set instance=%1
set database=%2

if /i "%instance%" ==
  "" set instance=.\SQLEXPRESS
if /i "%database%" == "" set database=UnitTest
```

An alternative approach is to use optional environment variables to specify the default values so that you never need to pass any arguments if you decide to tow the party line:

```
set instance=%personal_instance%
set database=%personal_database%

if /i "%instance%" == "" set instance=%1
if /i "%database%" == "" set database=%2

if /i "%instance%" == "" call :usage & exit /b 0
if /i "%database%" == "" call :usage & exit /b 0
```

Ultimate flexibility comes from combining the two approaches so that you can keep things really simple for the vast majority of use cases, but you still have the ability to override things at any point by supplying different values on the command line. In my experience database development is the one place where this has actually proved to be quite useful.

Developer sandbox set-up

If you’re going to allow (or need to accommodate) some element of personalised configuration then make sure it is optional. There is nothing more soul-destroying on your first day of a new job than being presented with a 30-page manual on what steps you need to take to configure your development machine so that you can actually do your job. And if you think I’m exaggerating by recounting a 30-page document – I’m not!

For me the steps required to get up and running on a new project should be as simple as:

1. Install version control software
2. Fetch development branch codebase
3. Build code and run tests

Step 3 might look like it should be split into two, but it’s not because that’s exactly what the build machine will be doing and so whatever it does I should be able to do, too. At this point I know that I can replicate what the build machine does and so I’m good to go.

Script composition

After you’ve created a few simple scripts it then becomes easier to see how you can combine them with other simple scripts to accomplish ever bigger tasks. Although it was never envisaged it would end up that way, many of the build processes I’ve been involved with in the past few years have ended up taking an imperative approach mainly due to the incremental approach of layering together many simple scripts to create a more complex process. The same goes for the deployment side as well.

For example the deployment process started out as two simple scripts that wrapped invoking `MSIEXEC.EXE` – one to install the package and one to uninstall it. The wrappers allowed me to handle the common error codes (e.g. 1605) that occur when the package is already installed/uninstalled. Once the NT services were written another simple wrapper around `SC.EXE` was created to start/stop them. These where then combined into a larger script that ensured the services were started/stopped before/after the packages installed/uninstalled. Add in another simple script to ensure any errant processes are terminated (`PSKILL.EXE`), another to copy the files from a known share to the local machine (`ROBOCOPY.EXE`) and finally a top-level script to act as the root for a Scheduled Task and you suddenly have an automated deployment process that even Heath Robinson would be proud of.

Scripting objects

On Windows there is a slight variation on this theme of driving command line tools using a very basic language; which is to drive ‘objects’ instead. If you consider the batch file language as the glue that binds together disparate console-style processes, then VBScript is the same simple glue that binds together objects exposed via COM. It might seem an expensive way to do business, but if your architecture is that classic pairing of a Visual Basic front-end and a C++ back-end then you’ve already done most of the heavy lifting. It might sound perverse but I’ve worked on a project where the entire build process was a library of VBScript files that were stitched together by using Windows Script Host (`.WSF`) files. It’s not something a sane person would consider doing today but 10–15 years ago it was Microsoft’s answer to the lack of scripting on Windows. That said, in a locked down production environment with legacy servers it might still be your only choice.

In a way that ideology still exists today in the guise of PowerShell. COM has been replaced by its modern heir – .Net – and the PowerShell language provides a much cleaner binding because .Net itself underpins it. Of course the pipeline model still exists too although it’s been ‘enhanced’ to flow objects instead of simple lines of text. Once again, if your core technology is already .Net you’ve done the heavy lifting and consuming your components via scripts is pretty easy. PowerShell may be Microsoft’s first choice, but the model works equally well with both F# and IronPython in the driving seat, although the latter seems to be sorely neglected these days.

Capturing pipelines

Of course none of this is going to replace the venerable UNIX pipeline which, despite its simple text based nature, lives on exactly because it’s simple and an army of programmers have created a wealth of small, well-focused tools that are designed to be composed to create something bigger than the sum of its parts.

Oftentimes I’ll need to do a little text processing and it ends up being a disposable one-liner. But other times I realise it actually might be useful to my team mates. There is a certain level of personal gratification in publishing your Byzantine one-liners to your fellow developers and if they only see the light of day once in a blue moon then the XKCD chart [1] rules. But if you think it’ll get a frequent work out then you might want to consider encapsulating it within a script for ease of use.

A few years ago I started work on a new project that sprouted the need to process line-based text files at every turn. This caused me to reacquaint myself with SED & AWK as part of a rediscovery of the classic pipeline. In fact you’ll find my delight documented in this very journal as part of the ‘Inspirational (P)articles’ series [3].

Listing 1

```
SED %1 … | SED … | SED … | SORT … | CUT … > "%TEMP%\lhs.csv"
SED %2 … | SED … | SED … | SORT … | CUT … > "%TEMP%\rhs.csv"
GUIDIFF "%TEMP%\lhs.csv" "%TEMP%\rhs.csv"
```

In one particular case after the system had gone live I started needing to compare some published CSV files between the development, test and production environments as part of the testing strategy. The files were not directly comparable so a little pre-processing was needed to first remove the noise. Old hands will no doubt recognise that a sprinkling of SED is one way to replace some variable string patterns with fixed (and therefore comparable) text:

```
SED "s/Timed-Out/ERROR/g" | SED "s/OutOfMemory/
ERROR/g"
```

Due to the non-deterministic nature of a SQL SELECT without an ORDER BY clause (which would have been an unnecessary burden on SQL Server) the file needed to be sorted. The best key to sort on was not the leftmost in the file which meant treating the file as having delimited fields:

```
SORT -t "|" -k 3,4
```

Finally, every time the file had new fields appended in a release they needed to be ignored when doing regression testing:

```
CUT -d "|" -f 1-24
```

The output was always fed directly to a GUI based diff tool in case there were differences to investigate and so structurally the script looked like Listing 1.

There was no reason (apart from the usual lack of time and a suitably privileged account) why that last step couldn't have used the normal DIFF tool and been automated to capture the differences in a report every morning.

Gall's Law [4] says that a complex system that works is invariably found to have evolved from a simple system that worked. Build, test and deployment processes in particular seem to have a habit of growing organically and judicious use of little scripts can be one way of slowly piecing together functionality by building on the existing set of tried and trusted tools. ■

References

[1] http://xkcd.com/1205

[2] http://chrisoldwood.blogspot.co.uk/2010/06/windows-batch-file-template.html

[3] *C Vu Journal* Vol 23 #1 (March 2011) and http://chrisoldwood.blogspot.co.uk/2010/11/reacquainting-myself-with-sed-awk.html

[4] http://en.wikipedia.org/wiki/Gall's_law

Management
Development
Security
Quality Assurance
coverity

MANAGEMENT
Increases visibility for better decision making, creates a predictable release process (on-schedule, on-budget), reduces costly support and production issues downstream, and provides consistent measurement of teams against common metrics to track improvement over time.

SECURITY
Creates a more efficient security audit process by eliminating a portion of security issues upfront, and helps focus testing and remediation efforts on the problems that require their expertise.

QUALITY ASSURANCE
Creates a more efficient QA testing process by receiving a higher quality build from the start, reduces wasted testing time due to buggy code, and focusses testing efforts on the problems that require their expertise.

DEVELOPMENT
Finds hard-to-spot defects in code, provides guidance to help fix problems fast – without requiring subject-matter expertise, reduces the time spent on re-work and de-bugging, and enhances skills by showing developers where they are most susceptible to error.

DEVELOPMENT TESTING =
faster time to market + reduced cost
+ greater customer satisfaction and brand equity
+ increased visibility and traceability
+ improved cross team collaboration + less risk

Development Testing

Development testing is an emerging category, including a set of processes and software, such as static analysis, designed to help developers, management, and the business easily find and fix quality and security problems early in the development cycle, as the code is being written, without impacting time to market, cost, or customer satisfaction.

Development testing augments traditional testing, including QA functional and performance testing and security audits, providing development teams with a quick and easy way to test their code for defects in a non-intrusive manner, so development stays focused on innovation, management gets visibility into problems early in the cycle to make better decisions, and the business continues to deliver high quality products to market for competitive advantage.

For more information, visit **www.coverity.com/development-testing**

coverity®

Got an event to plan?
Let Archer Yates Associates take the strain.

We are a leading, professionally recognised event management company. With over 30 years experience of organising and managing events, from conception right through to completion, on all scales from an executive board meeting to large annual conferences.

Let us help you create a memorable event that reflects and exceeds your targets and expectations. We can research, plan, deliver and evaluate every element from travel to destination, hotel and entertainment to technical production and booking management.

For a no obligation chat and to help you develop your ideas further, contact our friendly team on 01608 659900 or email Managing Director Julie Archer at Julie@archer-yates.co.uk

"Today's Solution to Tomorrow's Event"

Archer Yates Associates Ltd » Tel: +44 (0)1608 659900 » www.archer-yates.co.uk

aim ACCREDITED IN MEETINGS Member | The Association of British Professional Conference Organisers ABPCO Full Member | Green Tourism

The Ghost of a Codebase Past

Pete Goodliffe leads us down memory lane.

I will live in the Past, the Present, and the Future.
The Spirits of all Three shall strive within me.
I will not shut out the lessons that they teach!
~ Charles Dickens (A Christmas Carol)

Nostalgia isn't what it used to be. And neither is your old code. Who knows what functional gremlins and typographical demons lurk in your ancient handiwork? You thought it was perfect when you wrote it – but cast a critical eye over your old code and you'll inevitably bring to light all manner of code gotchas.

Programmers, as a breed, strive to move onwards. We love to learn new and exciting techniques, to face fresh challenges, and to solve more interesting problems. It's natural. Considering the rapid turnover in the job market, and the average duration of programming contracts, it's hardly surprising that very few software developers stick with the same codebase for a prolonged period of time.

But what does this do to the code we produce? What kind of attitude does it foster in our work? I maintain that exceptional programmers are determined more by their attitude to the code they write and the way they write it, than by the actual code itself.

The average programmer tends not to maintain their own code for too long. Rather than roll around in our own filth, we move on to new pastures and roll around in someone else's filth. Nice. We even tend to let our own 'pet projects' fall by the wayside as our interests evolve.

Of course, it's fun to complain about other people's poor code, but we easily forget how bad our own work was. And you'd never intentionally write bad code, would you?

Revisiting your old code can be an enlightening experience. It's like visiting an ageing, distant relative you don't see very often. You soon discover that you don't know them as well as you think. You've forgotten things about them, about their funny quirks and irritating ways. And you're surprised at how they've changed since you last saw them (perhaps, for the worse).

Looking back at old code you've produced, you might shudder for a number of reasons...

Presentation

Many languages permit artistic interpretation in the indentation layout of code. Even though some languages have a de-facto presentation style, there are still large gamut of layout issues which you may find yourself exploring over time. Which ones stick tends to depend on the conventions of your current project, or on your experiences after years of experimentation.

A classic example from the C++ programmer camp: many developers follow standard library layout:

```
class standard_style
{
  int variable_name;
  bool method_name();
};
```

and some have more Java-esque leanings:

```
class JavaStyle
{
  int variableName;
  bool methodName();
};
```

A simple difference, but it profoundly affects the code you work on in several ways.

Another example that, pertinent to my current workplace, is the layout of C++ member initialiser lists. We used to write something like this:

```
Foo::Foo(int param)
: member_one(1),
  member_two(param),
  member_three(42)
{
}
```

That's not too unfamiliar a style. However, we have recently switched to a style that places the comma separators at the beginning of the following line, thus:

```
Foo::Foo(int param)
: member_one(1)
, member_two(param)
, member_three(42)
{
}
```

There are a number of advantages to this style (it's easier to 'knock out' parts in the middle via preprocessor macros, or comments, for example). This scheme can be employed in a number of layout points (e.g. lists of all sorts of things: members, enumerations, lists of base classes, and more), providing a nice consistent style. There are also disadvantages: one of the major cited issues being that it's not as 'common' as the former layout style. IDEs' default auto-layout also tends to fight with this scheme.

I know over the years that my own presentation style has changed wildly, depending on the company I'm working for at the time.

As long as a style is employed consistently in your codebase, this is really a trivial concern and nothing to be embarrassed about.

The state of the art

Most languages have rapidly developed their in-built libraries. Over the years the Java libraries have grown from a few hundred helpful classes to a veritable plethora of classes, with different skews of the library depending on the Java deployment target. Over the many C# revisions the standard library has burgeoned. As languages grow their libraries accrete more features.

And as those libraries grow, some of the older parts become deprecated.

Such evolution (which is especially rapid early in a language's life) can unfortunately render your code anachronistic. Anyone reading your code for the first time might presume that you didn't understand the newer language/library features, when those features simply did not exist when the code was written.

For example, when C# added generics, the code you would have written like this:

```
ArrayList list = new ArrayList(); // untyped
list.Add("Foo");
list.Add(Int(3)); // oops!
```

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

with its inherent potential for bugs, would have become:

```
List<string> list = new List<string>();
list.Add("Foo");
list.Add("Bar");
```

There is a very similar Java example with surprisingly similar class names!

The state of the art moves much faster than your code. Especially your old, untended code.

The splendid new C++11 language features and library support has made much old C++ code look questionable. The introduction of a language-supported threading model and library renders third-party thread libraries (often implemented with rather questionable APIs) redundant. The introduction of lambdas removes the need for a lot of verbose hand-written 'trampoline' code. The range-based for helps remove a lot of syntactical trees so you can see the code's design 'wood'. Once you start using these facilities, returning to older code without them feels like a marked retrograde step.

Idioms

Each language, with its unique set of language constructs and library facilities, has a particular 'best practice' method of use. These are the idioms that experienced users adopt, the modes of use that have become honed and preferred over time.

These idioms are important. They are what experienced programmers expect to read; they are familiar shapes that enable you to focus on the overall code design rather than get bogged down in macro-level code lines. And they usually formalise patterns that avoid common mistakes or bugs.

It's perhaps most embarrassing to look back at old code, and see how un-idiomatic it is. If you now know more of the accepted idioms for the language you're working with, your old non-idiomatic code can look quite, quite wrong.

Many years ago, I worked with a team of C programmers moving (well, shuffling slowly) towards the (then) brave new world of C++. One of their initial additions to a new codebase was a max helper macro, shown below (do you know why we have the brackets in there?):

```
#define max(a,b) ((a)>(b)) ? (a) : (b))

void example()
{
int a = 3, b = 10;
int c = max(a, b);
}
```

After some time, someone revisited that early code and, knowing more about C++, realised how bad it was. They re-wrote it in the more idiomatic C++ below. This fixed some very subtle lurking bugs, as shown in Listing 1.

The original version also had another problem: the macro name clobbers a function name in the C++ standard library, with all sorts of unpleasant side-effects. This hints at a further problem: wheel reinvention. The best solution is to just use the built-in **std::max** function that always existed. It's obvious in hindsight:

```
// don't declare any max function

void even_better_example()
{
  int a = 3, b = 10;
  int c = std::max(a,b);
}
```

This is the kind of thing you'd cringe about now, if you came back to code like this. But you had no idea about it back in the day.

exceptional programmers are determined more by their attitude to the code they write and the way they write it, than by the actual code itself

Design decisions

Did I really write that in Perl? Did I really use such a simplistic sorting algorithm? Did I really write that by hand, rather than just using a built-in library function?

As you learn more, you realise that there are better ways of formulating your design in code. This is the voice of experience.

Bugs

Perhaps this is the reason that drags you back to an old codebase. Sometimes coming back with fresh eyes uncovers obvious problems that you missed at the time. After you've been bitten by certain classes of bug (often those that the common idioms steer you away from) you begin to naturally see potential bugs in the code. It's the programmer's sixth sense.

Conclusion

Looking back over your old code is like a code review for yourself. It's a valuable exercise to do; perhaps you should take a quick tour through some of your old work. Do you like the way you used to program? How much have you learnt?

Does this kind of thing actually matter? If your old code's not perfect but it works, should you do anything about it? Should you go back and 'adjust' the code? Probably not – if it ain't broke don't fix it. Often the code does not rot, unless the world changes around it (compiler versions break your old code, or the latest library version no longer let you compile).

It's important to appreciate how times have changed, how the programming world has moved on, and how your personal skills have improved over time. Finding old code that no longer feels 'right' to you is actually a Good Thing: it shows that you have learnt and improved. Perhaps you don't have the opportunity to revise it now, but knowing where you've come from helps to shape where you're going in your coding career.

Like the ghost of Christmas past, there are interesting cautionary lessons to be learnt from our old code if you take the time to look at it. ■

Questions

1. How does your old code shape up in the modern world? If it doesn't look too bad, does that mean that you haven't learnt anything new recently?
2. How long have you been working in your primary language? How many revisions of the language standard or built-in library have been introduced in that time? What languages features have been introduced that have shaped the style of the code you write?
3. Consider some of the common idioms you now naturally employ. How do they help you avoid errors?

Listing 1

```
template <typename T>
inline max(const T &a, const T&b)
{
  // Look mum! No brackets needed!
  return a > b ? a : b;
}

void better_example()
{
  int a = 3, b = 10;

  // this would have failed using the macro
  // because ++a would be evaluated twice
  int c = max(++a, b);
}
```

Code Critique Competition 82

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.

Last Issue's Code

"I am new to C++ and trying to write some objects to disk and read them back in. How can I get the pointer to the objects that are read back in?"

Where would you start with trying to help this newcomer?

The code is in Listings 1, 2, 3 and 4.

Listing 1

```
/*
 * Bike.h
 */

#ifndef BIKE_H_
#define BIKE_H_

#include <iostream>
#include <string>
#include <vector>
#include <iterator>
#include <algorithm>
#include <iomanip>
#include <ios>

class Bike {
  Bike* address;  // Pointer to Bike object
  std::string name;
  double price;
  std::string make;

public:
  //Bike(); // eliminate to avoid ambiguity
  Bike(Bike* a, const std::string& n =
    "unknown", double p=0.01,
    const std::string& m="garage") :
    address(a), name(n), price(p), make(m){}
  virtual ~Bike();

  inline std::string getName(){return name;}
  inline double getPrice(){return price;}
  inline std::string getMake(){return make;}
  inline Bike* getAddress(){return address;}

  static void writeToDisk(
    std::vector<Bike> &v);
  static void readFromDisk(std::string);
  static void splitSubstring(std::string);
  static void restoreObject(
    std::vector<std::string> &);
};

std::ostream& operator << (std::ostream& os,
  Bike &b);

#endif /* BIKE */
```

Listing 2

```
/*
 * Bike.cpp
 */
#include "Bike.h"
//Bike::Bike() {} // TODO Auto-generated stub
Bike::~Bike() {}  // TODO Auto-generated stub
std::ostream& operator << (std::ostream& os,
  Bike &m){
  os << std::left << std::setw(10)
     << m.getAddress() << "\t"
     << m.getName() << "\t"
     << m.getPrice() << "\t" << m.getMake();
  return os;
}
```

Listing 3

```
/*
 * file_io.cpp
 */
#include "Bike.h"
#include <fstream>
#include <iomanip>
#include <iostream>
#include <iterator>
#include <vector>
#include <cstring>
#include <sstream>
#include <algorithm>
// Write objects to disk
void Bike::writeToDisk(std::vector<Bike> &v){
std::ofstream out_2("bike_2.dat");
for (auto b:v){
  out_2 << b.getAddress() << ':'
      << b.getName()
      << ':'<< b.getPrice() << ':'
      << b.getMake()  << std::endl;
  }
out_2.close();
}
//-----------------------------------------------
//Read from disk into vector and make objects
void Bike::readFromDisk(
  std::string bdat) // "bike_2.dat"
{
  std::cout << "\nStart reading: \n";
  std::vector<char> v2;
  std::ifstream in(bdat);
   copy(std::istreambuf_iterator<char>(in),
       std::istreambuf_iterator<char>(),
       std::back_inserter(v2));
in.close();
```

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 3 (cont'd)

```
for(auto a:v2){
   std::cout << a;  // Debug output
   }
std::string s2(&(v2[0])); // Vector in String
std::cout << "\nExtract members:\n";
while (!s2.empty()){
  //  objects separated by \n
  size_t posObj = s2.find_first_of('\n');
  std::string substr = s2.substr(0,posObj);
  s2=s2.substr(posObj+1);
  splitSubstring(substr);
  }
}
void Bike::splitSubstring (std::string t){
  // Save the address and the members in v3
std::vector<std::string> v3{(4)};
size_t posM; // [in substring]
int i;
for (i=0; i<4; i++){
  posM = t.find_first_of(':');
  v3[i] = t.substr(0,posM);
  if (posM==std::string::npos) break;
  t=t.substr(posM+1);
  }
for(auto member:v3){
  std::cout << std::setw(10) << std::left
    << member << "    \t";}
  restoreObject(v3);
  std::cout << std::endl;
  v3.clear();
}
void Bike::restoreObject
   (std::vector<std::string> &v3){
  Bike* target;  // I want the object here ...
  double p;
  std::stringstream ss(v3[2]);
  ss >> p;
  Bike dummy{&(dummy),v3[1], p, v3[3]};
  target =  &(dummy);
  std::cout << "\nRestore: " << *target
    << std::endl;
}
```

Listing 4

```
/*
 * main_program.cpp
 */

#include "Bike.h"
#include <fstream>
#include <iomanip>
#include <iostream>
#include <iterator>
#include <vector>
#include <cstring>
#include <sstream>
#include <algorithm>

int main(){
  std::cout << "start\n";
  std::vector<Bike> v;
  Bike thruxton{&(thruxton), "Thruxton",
    100.00 , "Triumph"};
  Bike sanya{&(sanya)};
  Bike camino{&(camino), "Camino   ",
    150.00, "Honda"};
  Bike vespa{&(vespa), "Vespa    ",
    295.00, "Piaggio"};

  v.push_back(thruxton);
  v.push_back(sanya);
  v.push_back(camino);
  v.push_back(vespa);

  for(Bike b:v) std::cout << b << std::endl;
  // using overloaded << operator

  Bike::writeToDisk(v);
  // restore objects
  Bike::readFromDisk("bike_2.dat");
  // where are the restored objects??
  return 0;
}
```

Critiques

Paweł Zakrzewski <pawel@zakrzewski.cc>

I would wager that the person asking this question is new to programming in general. To get a pointer – or any value in fact – from a function, one should return it, write it to a reference parameter of said function or write it to a global variable (throwing an exception with that value is also possible, but that's plain abuse of language features that are meant for something else). The most natural way is to return them, so that's what I would suggest. That being said, there are numerous things wrong with the proposed program, and lack of understanding of the way functions work is but one of them. I'll try to list the issues in order of their severity, beginning with program-breaking ones, followed by runtime performance issues and finishing with some comparatively minor annoyances.

The way serialization is implemented gives away author's lack of foresight and experience. Not only the methods deal with non class-specific code, but also they are too specific. Adding network or database support would require a pair of methods each and if a class **`Car`** was to be created, no code could be reused. The best way to implement serialization is not implementing it at all and using an already existing well designed and tested solution, for example Boost.Serialization from Boost libraries. If that is impossible, only two methods should be created: one for serialization, one for deserialization, and they should be medium-agnostic, most likely reading and writing from/to a stream.

When handling disk I/O the author forgets to check whether the stream is open after construction. **`std::ifstream::is_open`** and **`std::ofstream::is_open`** should be called for input and output respectively. From a performance point of view, in case of bigger files, **`v2`** should reserve memory for all of the file contents, otherwise multiple unnecessary reallocations will occur. To get the file size, **`std::ifstream::seekg()`** and **`std::ifstream::tellg()`** functions should be used.

```
std::vector<char> v2;
std::ifstream in(bdat);
if(!in.is_open()) { /* error handling */ }
in.seekg(0, std::ios::end);
v2.reserve(in.tellg());
in.seekg(0, std::ios::beg);
copy(...); // as in the example
```

In the same function, a **`std::string s2`** is created from **`v2`**, using a constructor that takes a null-terminated character string. The parentheses around **`v2[0]`** are superfluous, but don't change the meaning. If **`v2`** contains null characters, which, given the **`Bike::writeToDisk()`** function, is unlikely, not all data will be copied. Otherwise, **`v2`** will be accessed outside of its boundaries, which is undefined behaviour. In my opinion, **`v2`** is unnecessary and the string should be used in its place from the beginning of the function.

Just a few lines below, there's another instance of not checking the returned value.

`std::string::find_first_of()` returns **`std::string::npos`** when it doesn't find the required character. The current code doesn't check for that and if that happens, **`posObj+1`** in **`s2 =`**

s2.substr(posObj+1) will evaluate to 0 (if two's complement is used) or **std::numeric_limits<size_t> ::max()** (in one's complement), which will end with an infinite loop (**s2** will never change, since **std::string::substr(0)** returns a copy of the string) or **std::out_of_range** being thrown respectively.

That loop is also a cause for numerous unnecessary copies. Instead of assigning **s2**'s substring to **s2**, **s2.erase(0,posObj)** would be more effective. It's also possible to iterate over **s2**'s lines without modifying it at all:

```
for(auto it = std::begin(s2),
         e = std::end(s2); it != e;){
  auto found = std::find(it, e, '\n');
  // returns end of string iterator if nothing
  // is found
  std::string substr{it, found};
  splitSubstring(substr); // this name should
                          // be changed
  it = found;
}
```

The naming scheme used leaves a lot to be desired. For some reason function **Bike::splitSubstring** calls **Bike:: restoreObject**, which is supposed to deserialize a single **Bike** object. This name is extremely misleading and any code created later may break the program. What's more, it's most likely not supposed to be used outside of the class, yet it is in its public section.

Another thing worth noting is the **Bike::address** pointer that is used to keep address of the instance that owns it. While not strictly wrong from the language point of view, it only serves as distraction and possible ambiguity for anyone reading the code, as the same result can be achieved using **operator&** or **std::addressof** on **Bike** objects and this pointer in **Bike**'s methods. What's more, the implicitly created **copy** constructor will copy other instance's address, which may break the program and cause undefined behaviour if dereferenced when a vector of **Bikes** is reallocated. My suggestion is to remove that pointer and its getter.

Including commented out declaration and definition of a default constructor doesn't eliminate it. In this case, the desired effect is achieved by supplying a non-default constructor. If the author wants to be really explicit about removing the default constructor, they can do so in two ways: compatible with C++98 – declaring a private default constructor without defining it, and, since C++11, declaring **Bike() = delete**. Since C++11 is already used, I would use the latter option, because its intent is clear to anyone speaking English, while the former solution requires the reader to know that particular C++ idiom.

The author seems to have copied some code they saw elsewhere without understanding reasons for using various constructs. A public virtual destructor, usually in conjunction with the base being abstract, should only be used when there is dynamic polymorphism involved. Otherwise, the additional flexibility is not used, but the performance cost doesn't go away. In this case, it is not necessary.

The next thing to change are getters. I already suggested removing **getAddress**, along with the address variable, but the others aren't exactly good either. First of all, they shouldn't require a non-const **Bike** instance, as they don't do anything but read – they should be **const**. Secondly, returning **std::string** – and, in fact, most non built-in types – by **copy** is wasteful, **const** reference should be applied. The rule of thumb when one doesn't want to modify the original value is: if the type is not a keyword or a **typedef** of a keyword, return it by **const** reference.

The same thing goes for function parameters that are supposed to be read only, although, since C++11, it may be a little more complicated. For example, **Bike::writeToDisk** (assuming it isn't scrapped) should take **std::vector<Bike> const&** parameter. But, if at any point it copied from such variable while abandoning the parameter (as often happens in class constructors, for example), **const** reference may incur an unnecessary copy, as it indeed does in **Bike**'s constructor, that, funnily enough, takes **const** references to strings.

Let's consider what happens to parameter **n** of class **Bike** constructor, when called like it's called in `main_program.cpp`: **Bike(&(thruxton), "Thruxton", 100.00, "Triumph")**. First, a **std::string** instance is initialized with **"Thruxton"**, then it is passed by **const** reference to the constructor, and then the **const** reference is passed to **Bike::name**'s (which is a **std::string**) constructor, but since it sees a **const** reference it has to make a copy, instead of a move.

C++11 introduced **rvalue** references, which are guaranteed to be possible to be moved from, but to avoid copying when not necessary, the function needs to be overloaded for every parameter that may be taken by rvalue reference. Unfortunately, the number of overloads rises exponentially with number of parameters being overloaded on. It is possible to use perfect forwarding with **std::forward**, but that has the unfortunate effect of moving implementation to the header file.

The solution is to take such argument by copy. This may come as a shock to some people, but let's consider **Bike**'s constructor again, assuming that it takes **n** by copy and then initializes name with **std::move(n)**. If called exactly like above, **n** is initialized with **"Thruxton"**, then it is moved into **Bike::name** with no unnecessary copies. On the other hand, if the constructor was called like this:

```
std::string thruxton_name{"Thruxton"};
Bike thruxton{&(thruxton), thruxton_name,
  100.00, "Triumph"};
```

thruxton_name would be copied to parameter **n**, which would then be moved to **Bike::name**. That's exactly one move more expensive (and moves are very cheap) than the constructor taking **const** reference to string, but at the same time much more flexible.

Ranged for loop (introduced in C++11) is used throughout the code. In all instances, the loop variable is taken by copy. It probably may be excused* for the debug output in **Bike::readFromDisk**, but in other cases, **const** reference should be taken, for example

```
for(Bike const& b: v)
```

or

```
for(auto const& member : v3)
```

* In that case, taking char by copy is okay, but the whole loop isn't necessary, because it produces **v2.size()** of **operator<<** calls, when a simple **std::cout.write(&v2[0],v2.size())** would suffice. If **s2** was used as suggested above, this would be as simple as calling **std::cout << s2;**

In the **main()** function, if the **Bike::address** variable was removed as suggested above, vector **v** could be populated using **std::vector<Bike>::emplace_back()** initialized with an initializer list. In this case, the change wouldn't be noticeable, but it's a good habit to avoid premature pessimizations.

Although it won't matter in such a small program, putting unnecessary includes is a bad practice, especially in other header files. It slows down the compilation and, should any of included files change, the including file is also treated as changed, which forces recompilation. For example, `Bike.h` doesn't seem to need anything but **<string>**, **<iosfwd>** and **<vector>**.

Stray thoughts:

- Most projects use their own namespaces to avoid name collisions. It may be more important in libraries, though.
- Most projects I've encountered prepend or append **"_"**, **"m_"** or **"p_"** to class instance variable names in order to make it clear what they are. It is far from mandatory, but still worth consideration.
- Consider using integral variables for storing price. They differ much less between platforms and are not susceptible to imperfect rounding.

Balog Pál <pasa@lib.hu>

We're facing a lot of really awful code, where to start indeed. In real life if review discovers showstopper problems, it stops aiming for completeness – just the next pass is scheduled after the author addresses the first bunch. And the code is cleaned/refined in iterations.

The first thing that jumps out is a `Bike` pointer in the `Bike` object. Pointer to the same types are pretty rare, mostly appear in C code where payload is mixed with container handling and the pointers serve to build a list or a tree. As bike suggests no reason, let's look what address is used for. Just taking a small detour to comments. As we have one here at the perfect spot. With the perfect content too – for an anti-comment that is.

I call *anti-comment* the thing that states the obvious. Tells me what I see from the code. Without providing any information. Sure, address is a pointer to a `Bike` object. I know that from the type. What I want to know, is what kind of bike it is intended to point, why, when, can it be `NULL`, do I have to allocate it, do I have to free it, how lifetime of the pointed object relates to my instance, and stuff like that. When I see a pointer in a code, be it member of a structure or parameter of a function all those questions are formed. And most code is just silent, the author did not think to drop a comment answering them. An anti-comment adds insult to the injury.

Back to chasing *address*: it is set from parameter passed in `ctor`, stored and only accessed in `getAddress()`. The class does not use it. `getAddress()` is used in `op<<` and `writeToDisk` just dumping the content. Constructor of `Bike` is found in `restoreObject` and `main`. In all cases the address of the currently created instance is passed.

This is kinda relief in one way: to fix the code we send the whole thing to `/dev/null` and drink a pint on good riddance. The more difficult task is to explain the client the problem without being able to ask him first how he got the idea in the first place.

In C++ when we create objects the system manages the address. We do not mess with the Zohan. Neither do we need to; in member function if we're interested in our address it sits conveniently in the this pointer and outside we can use the `&` operator. Once the object is created, not before or the middle of it.

In member functions usage of this is implicit: when we refer to the name of a member, it's like we've written `this->name`. Yes, I deliberately kept placement new secret, and allowed uses of `this` on half-constructed object, that is advanced subject half year ahead.

With the pointer issue gone `restoreObject` can simply assign the input strings to `name` and `make` and stream directly to `price`. Or construct a local `Bike` object just as in original attempt from the three parts and assign it to `*this` after some validation. Related important topics here are 'All input is evil' and 'strong exception guarantee'. But I leave those for the second review, at which point the class may have gained some actual rules and invariant, and the application some error handling policy. Currently `Bike` is just a dump of data and requirements are not stated. Simple assignment fits the basic guarantee and the client is not present to answer 'what behavior you expect if the middle member does not look like a double'. Oh yeah, to work we remember to remove `static` in declaration.

We resume at class `Bike` in the header that now starts with `name`. Using string and double looks fair, I'm not really happy with `make` that gets initialized with some kind of brand, but I'm not versed in biking, it may be the proper lingo. The members are private that is good.

We arrive at the `ctor`. The commented `ctor` can now be removed as the other is good for the default case. We just insert the `explicit` keyword. Ctor uses init list and nothing in body – perfect. Without the client I just assume the defaults make sense in use and content.

Without `explicit` any string sitting around would be happy to convert to `Bike`, that doesn't look sensible, just serves as a time-bomb.

We see a virtual dtor that does nothing – so we send it after address to oblivion.

We use virtual dtor in classes meant as base class. A usable base class likely have virtual functions, and ultimately documentation on usage and hierarchy rules. We might re-insert this dtor later on when time is ripe.

Next come accessors for all members, not very nice but without setters not very harmful either. We cut `inline` at front and add `const` after `()` before moving on.

The only sensible use of `inline` keyword is for functions defined in a header at namespace level, to prevent the compiler creating multiple bodies. A function defined in class is implicitly inline so stating it is just noise.

`const` should have been the default but is not for compatibility reason. So remember to add it everywhere. Especially at end of member functions unless they are meant to change the state of the object.

Then come some strange public static functions. They all seem to serve serialization. Which is a topic that could easily make up a whole book. And best be left to existing framework like Boost::serialization. Sure we could implement the I/O of three puny members here, but in an application we normally face many objects with similar requirement and that asks for some general structure or framework.

I don't aim for a complete solution here just show guidelines to cut the knot. For sensible work each class only supplies one pair of functions that save and load one instance taking an abstract archive stream as parameter. The stream knows whether it's connected to disk file or console or TCP/IP or whatever. Dealing with collections, rather than individual objects is also handled outside. So `WriteToDisk` is ill-conceived. Instead we need a non-static member like `void Save( OARCHIVE&) const;` Without a framework we can start using the current implementation, make OARCHIVE `std::ostream& out`, and use the central piece of the (existing) implementation. Certainly that function moves to `bike.cpp`. The outer section can become a simple function moved to `main_program` for now. As it is arbitrary test-like code anyway using wired filename is not a problem, and error handling will be added later. The `for` loop calls just `Save()`, or we can conveniently replace it with `for_each` now.

The other function we need is the counterpart, that looks like `void Load( IARCHIVE&);` yeah, with `const` intentionally missing – also implemented in `bike.cpp`, and can be done by recombining the code in the three other functions. Or scrapping the content for good and just do the same as `Save` in the opposite direction. The base requirement is that the pair is in sync: `Load` must extract from the stream whatever `Save` placed there for the object. And restore to the same state. Store format can be chosen for convenience of implementation, say writing length info on save so load need not hunt for delimiters. It's usual to expect the stream be at correct point and having properly formatted content, and throw exceptions on any discrepancy. I leave the implementation as exercise until the next review. The outer code opening the file and messing with vector again moves to main file and we scrap `file_io` for good.

We're left with `<<` operator that misses a `const` in the signature as it has no business messing up the instance. It's implemented using the public getters that is good. I would not object if it was friend and used the members directly as long as it has read-only privileges. If getters appear to be there only for this reason it is good idea to rearrange that way and cut them.

The code in `main` now can be simpler, just using `push_back` on the fly, or even just an init list for the whole vector. More anti-comments can be cut and the question at the end must be answerable by anyone with Little Prince's mind able to look through the box. If there's still remaining confusion I can hint the load implementation that in the loop we have `Bike` instance we call `.Load` for and use `push_back` for the vector that is likely a parameter or return value. Or if collection I/O code wrote out `.size()`, load starts resizing the vector and can `.Load` to `v[i]` conveniently. And the vector itself is either param passed in by ref or is returned.

Some relatively minor issues:

- stuff outside include guards of a header
- excess includes in `.h`
- some genuinely hostile formatting of source.

Commentary

It is often hard to know where to start when reviewing the code of someone new to a given language: which is the most useful correction to make first? It is in cases like this that having more regular involvement with the developing program can be useful – being presented with a completed project means changes are likely be harder to make.

I think one of the key mistakes in this critique is the presence of the member data `Bike* address;` I think this reveals a lack of understanding of the idea of an ‘object’ in C++. If the programmer can be shown how to replace the use of address with `this` or the `&` operator as appropriate it may help them to understand something of the mechanism of object oriented programming.

The second place I would like to focus on is the streaming in of a new object: again, the root problem seems to be in understanding what is needed to create a new object. While a framework such as Boost.Serialization is a good solution in general I’m not sure whether it makes it more or less easy for the programmer in this example to understand what is involved with creating a new object.

It is also good practice to be careful with routine managing I/O that errors are handled properly. There are several possible classes of problems ranging from failure to read data from the file to attempts to handle input data of the wrong format – for example saved by a different version of the same program. In this day and age you must also think about the potential for exploits caused by reading deliberately invalid data.

The Winner of CC 81

Both critiques picked up many of the faults in the original code and both suggested a hierarchy of problems to be addressed. I think both did a good job of pointing the programmer in the right direction, given the sorts of problems that the code demonstrates. However, Paweł additionally picked up the lack of error handling as a fault deserving correction. I think that this is very important – the lack of good error handling is a congenital problem in our industry and I think it is good to address that early on. So I have awarded the prize for this critique to him.

Code Critique 81

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

This is a slightly different critique from the usual. The following code was presented to a number of people at the recent ISO C++ standards meeting and we were asked to work out what it would print. Please do so yourself – and *then* compile and run it. The critique part is to reflect on what happens, why it happens, and how to deal with it.”

(My thanks to Alan Talbot for this interesting example.)

The code is in Listing 5.

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://www.accu.org/journals/). This particularly helps overseas members who typically get the magazine much later than members in the UK and Europe.

Listing 5

```
#include <iostream>

int x;

struct i
{
  i() { x = 0; }
  i(int i) { x = i; }
};

class l
{
public:
  l(int i) : x(i) {}
  void load() {
    i(x);
  }
private:
  int x;
};

int main()
{
  l l(42);
  l.load();
  std::cout << x << std::endl;
}
```

DeveloperFocus

Effective C++11 programming with Scott Meyers

Software developers familiar with the fundamentals of C++11 are ready to advance from simply knowing C++11 to having a full understanding of how to effectively apply it. This 2 day course, based on Scott Meyers´s forthcoming ***Effective C++11***, highlights the insights and best practices of the most accomplished C++11 programmers. If you’re ready to advance your knowledge and strengthen your confidence using C++ 11 then visit the DeveloperFocus website for more information and booking.

DeveloperFocus aim to be one of the world’s leading providers of high-end training for developers and leaders. We strive to offer the developer community the best training as well as the hottest names and trends. Our philosophy is to provide ‘practical, applicable knowledge’ by providing the highest quality training with the best instructors and excellent customer service.

www.developerfocus.com
info@developerfocus.com
0843 523 5765

Standards Report

Mark Radford examines a knotty issue of lifetime facing the C++ standards committee.

After such eager anticipation, the Bristol ISO C++ meeting seems a long time ago now. The dust has now settled, and the post Bristol mailing has been published. You can find all the papers here: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/#mailing2013-05.

If you look down the list of papers you might notice that four of them have the word ‘async’ in their titles. Also, in addition to those four, the word ‘future’ appears in the title of one more paper, and there is a further paper on shared locking. There may be more, but these six papers are the ones I’ve noticed looking down the list. Therefore, it doesn’t take a genius to work out that concurrency is still a hot topic of discussion in the C++ standardisation process!

One specific subject of some controversy, is whether **std::future** should be changed so that its destructor never blocks. The way it is currently specified, a **future**’s destructor never blocks except when it is returned from **std::async**. When a future is returned from **std::async**, unless its **wait()** or **get()** member functions have already been called, its destructor will block until the thread behind the asynchronous operation has joined. In N3630, Herb Sutter *et al* present arguments that a **future**’s destructor should never block. In N3679, Hans Boehm takes the opposite side of the debate, arguing that a **future**’s destructor must wait for the thread to join. The problem is that with a **future**’s destructor blocking, surprising behaviour can result. The other problem is that, if the **future**’s destructor doesn’t block, it can be very easy to let undefined behaviour slip into the code.

To illustrate how the behaviour can be surprising, I’ll give a simple example from N3630:

```
{
  async( launch::async, []{ f(); } );
  async( launch::async, []{ g(); } );
}
```

In this example, both calls to **async()** execute sequentially i.e. there is no concurrency at all. This is because the **future** objects returned from **async()** are temporaries, and therefore go out of scope at the end of the statement. Therefore, at the end of each statement the **future**’s destructor is executed and blocks until the underlying thread joins.

Another problem is illustrated by another example from N3630:

```
void func() {
  future<int> f = start_some_work();
  /*... more code that doesn't f.get() or
    f.wait(), and performs no other
    synchronization ... */
}
```

The question is: does **f** block when it goes out of scope? The problem here is that the user would (effectively) have to know if the implementation of **start_some_work()** calls **async()**, or uses some other mechanism to achieve concurrency (for example, it could just launch a thread using **std::thread**). In passing, note also that there is a further encapsulation issue here: say **start_some_work()** uses **async()**, but then is later changed to use another concurrency mechanism, there would be a silent change to the behaviour of the client code! More to the point, the options for changing the implementation are seriously limited.

One of the main arguments for **future** behaving the way it currently does is the possibility of the asynchronous operation using out of scope variables or dangling references. This is the simplest example (based on one given in N3630):

```
{
  int i=0;
  std::future<int> f = async([&]{
    i=42; return i;});
}
```

If **f**’s destructor didn’t wait for the underlying thread to join, the variable **i** might not still be in scope by the time **i=42** executes. This seems like a compelling argument for the status quo, except that (as Herb *et al* point out) the problem exists anyway:

```
std::future<int> f;
{
  int i=0;
  f = async([&]{ i=42; return i;});
}
```

Or even:

```
{
  int i=0;
  std::shared_future<int> f = async([&]{
     i=42; return i;});
}
```

Because **std::shared_future**’s destructor doesn’t block.

In light of the above two examples, the argument for changing the behaviour of **future**’s destructor does seem quite compelling. However, it does constitute a silent run time change to the behaviour of existing code which (arguably) makes the solution worse than the problem. Note that changing the return type of **std::async** in the hope that recompiling will catch the problem doesn’t work: much C++11 code is likely to use **auto** to declare the returned **future** (so the code would recompile and the problem would still be at run time). Herb Sutter has also submitted a proposal (N3637) to resolve the problems: N3637 proposes that **future**’s destructor (as well as **shared_future**’s destructor) should not block, and that **waiting_future** and **shared_waiting_future** (which would block in their destructors) should be added to the standard library. However this still does not solve the problem of causing existing code to break silently at run time.

N3637 was rejected at Bristol to allow time for further reflection. ■

MARK RADFORD

Mark Radford has been developing software for twenty-five years, and has been a member of the BSI C++ Panel for fourteen of them. His interests are mainly in C++, C# and Python. He can be contacted at mark@twonine.co.uk

while (you care about code)
{
   read (cvu && overload);
}
do(it);
because good code matters
ACCU
www.accu.org
PROFESSIONALISM IN PROGRAMMING

ACCU Conference 2013

Anna-Jayne Metcalfe shares her conference experience.

The annual ACCU Conference is an important event in the organisation's calendar for several reasons. It provides an opportunity for technically minded people to get together to discuss what's interesting in the world of software development, and to learn what other people are interested in. It's a place where vendors can publicise and demonstrate their products to an audience of like-minded and knowledgeable potential customers. It provides a platform for people to present ideas and techniques, an important part of which is the opportunity for people who have never presented before to take part, and learn from others who are adept at it. Above all, it's a social event where people who normally communicate by electronic means can meet face-to-face, and discuss issues with recognised experts in many fields associated with software and programming.

ACCU 2013 was an event where you could talk C++ with Bjarne Stroustrup (and many others), software testing with Steve Freeman and Brian Marick, SOA, Java and C++ with Nicolai Josuttis, managing software projects and quality with Tom Gilb, and what you plan to do with that Raspberry Pi with Eben Upton. I wonder who you'll be able to meet next year?

This year's ACCU Conference was held at the Marriott Hotel Bristol on 9th–13th April. That in itself was rather noteworthy, as for as long as we have been going to this conference (since 2007, amazingly), it has been held in Oxford, so Bristol was new territory for us.

The ACCU 2013 foyer and registration desk

The main hallway at ACCU 2013

Beth and I arrived at the conference venue on Monday afternoon, and started setting up on Tuesday. The venue is a good one (and most importantly: one which gives the conference space to grow again!), but obviously after so many years at the previous venue in Oxford, there were lots of little things to relearn and adapt to.

Almost all of the usual faces were there, so it was good to see everyone again and catch up with what they've been up to over the last few months.

The Riverblade stand at ACCU 2013 just after we'd finished setting up

One minor change for me is that I've finally gone tablet and was taking notes this year in Evernote on a Nexus 7 3G rather than a netbook. The netbook is easier to type on, but being essentially a big phone the Nexus proved to be much more spontaneous in nature (though the lack of undo/redo and not being able to plug in a USB key without adaptors and rooting the device proved to be a bit of a pain). But you can't have everything, right?

Our stand (bottom left) was at the far end of the long hallway in the second photo above, opposite one of the entrances to the main conference room.

If knowledge reuse is your thing, John Jagger's Bletchley Park fundraising bookstand was the place to be (Upper photo courtesy of Kate Gregory)

Like all of the sponsors we had a tea/coffee point (with expresso machine!!!) next to us, and during the breaks food was served directly from tables directly opposite all along the hallway (so if one catering point was too busy, you could just find another one without any trouble).

In practice, the layout worked pretty well (catering in particular was less of a bottleneck than at the previous venue), although the hallway did get a little crowded at times.

Eben Upton of the Raspberry Pi foundation kicked things off in fine style on Wednesday morning with an entertaining keynote during which he described the concepts behind the Raspberry Pi, and some of the more surprising things people are doing with it.

Shared family computers are not compatible with kids learning to program – installing Python would disturb the viruses

~ Eben Upton

Eben Upton's Raspberry Pi keynote

Although we saw him give a similar talk at least year's Agile on the Beach [1], it was a very enlightening and entertaining keynote.

On Wednesday afternoon I was among those who

ANNA-JAYNE METCALFE

Anna is a C++ developer who opted out of commuting by starting a software company. She's worked on projects ranging from software quality tools to automated test, subsea navigation, digital broadcasting and chemical detection systems. She may be contacted at anna@riverblade.co.uk.

As ever, Blackwells were offering tempting discounts for ACCU 2013 delegates

did a 5 minute stand-up during Pete Goodliffe's 'Becoming a Better Programmer' session.

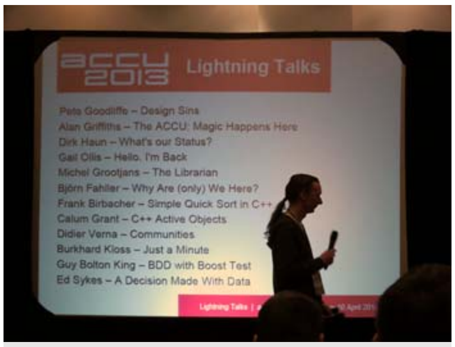
My talk was on the subject of continuous refactoring, and titled 'If it ain't broke, Do fix it' (I'll write this talk up as a blog post sometime). As luck would have it, I was up last and by the time my turn came I was really nervous and my delivery was all over the place.

I honestly thought I'd completely fluffed it – so when during his 'Cheating Decline: Acting now to let you program well for a really long time' keynote the following morning Brian Marick said that of all of the presenters in that session I was the one who nailed it for him – and them called me up front to present me with a book (*Beyond the Brain: How Body and Environment Shape Animal and Human Minds* [2]) – you could have knocked me over with a feather (fortunately nobody did, and I made it back to my seat without incident).

becoming a better programmer
PETE GOODLIFFE pete@goodliffe.net @petegoodliffe
ALL★ CAST
Pete Goodliffe's 'Becoming a Better Programmer' session on Wednesday afternoon

The big draw this year was of course Bjarne Strousrup's keynote 'C++ 11 The Future is Here', and his follow up session on the roadmap for future versions of the standard – notably C++ 14 and C++ 17.

Bjarne actually stopped by our stand at one point, but rather than talking about our stuff to any great extent we spent most of our conversation talking about arcade games (we have a Space Invaders keyboard on one of the machines on our stand).



ACCU 2013 Lightning Talks

Pete Goodliffe – Design Sins
Alan Griffiths – The ACCU: Magic Happens Here
Dirk Haun – What's our Status?
Gail Ollis – Hello, I'm Back
Michel Grootjans – The Librarian
Björn Fahller – Why Are (only) We Here?
Frank Birbacher – Simple Quick Sort in C++
Calum Grant – C++ Active Objects
Didier Verna – Communities
Burkhard Kloss – Just a Minute
Guy Bolton King – BDD with Boost Test
Ed Sykes – A Decision Made With Data

ACCU 2013 LIghtning Talks

Still, to have the opportunity to chat with the author of the C++ programming language about your stuff – however briefly – is pretty damn cool no matter how you look at it!

Thursday afternoon was Olve Maudal's famous (and appropriately titled) C++ 11 Pub Quiz – which of course took place in the bar. The fact that we were all offered a free pint (sadly, bottled as the bar did not have any real ale) made the atmosphere as relaxed and fun as I'm sure you can imagine.

Bjarne Stroustrup presenting 'C++11: The Future is Here' at ACCU 2013 (Photo courtesy of Dmitry Kandalov)

Although some of the questions were obscure (to say the least) the answers were at least subject to a well lubricated peer review!

While on the subject of liquid non-caffeinated refreshments, mention must be made of the astonishing Bloomberg Lounge which featured at this year's conference. For some reason (beer, perhaps?) I didn't take any pictures in there, but if you imagine a fairly bright but relaxed area with bar, retro gaming machines and a pool table you aren't far off the mark.

Olve Maudal's 'C++11 Pub Quiz' at ACCU 2013

Bloomberg were obviously showcasing their trading displays, which as a user interface developer I found quite interesting in themselves. Astonishingly, Bloomberg also sponsored a free bar for much of the week – a gesture many delegates took good advantage of as I am sure you can imagine!

Ook: design principles

A programming language should be writable and readable by orang-utans.

To this end, the syntax should be simple, easy to remember, and not mention the word "monkey".

Bananas are good.

Ook: Commands

Michel Grootjans presents 'Ook: A programming language designed for orang-utans' at ACCU 2013

The lightning talks this year were a blast, but for my money the highlight was a talk by Michel Grootjans on the 'Ook' programming language:

Seriously, some of the stuff you encounter at tech conferences is just comedy gold.

On Thursday evening the conference hosted Bristol Girl Geeks, at which Astrid Byro, Francis Buontempo, Kate Gregory and I all gave short talks. Mine was titled 'All alone in a sea of hairdressers',

Inspiration [P]article

Frances Buontempo shares a story about how engaging with someone can be fun and rewarding.

Having recently been interviewed by Gail Ollis for her PhD [1], I was left thinking about how easy it is to be negative when thinking about programming and, feeling we should make an effort to be positive just occasionally, realised it had been a long time since *CVu* had had an inspiration particle.

Recently some work experience pupils were sent round to meet us and other teams. I observed them coming towards us and what happened with the other teams. People were tending to just talk *at* them, explaining what they did day to day. I don't know about you, but having people talk at you for hours can make it hard to pay attention. It's often more interesting to actively engage with things. Having had a brief look at Jon Jagger's cyber-dojo [2] at an ACCU London meeting, this put an idea in my head.

When two of the pupils got sent to me, I asked if they had programmed before, and they said, "No". They admitted they had used some application which they could give instructions to, to get a taste of programming but that was it. We therefore spent about an hour coding 'FizzBuzz' in python using the cyber-dojo. It was marvelous just being able to try it out, without having to set anything up, and they could go home and have another go easily if they wanted. It was so exciting watching them go from afraid to braver, and more willing to try things. You could see a smile forming as they both had a go at typing and seeing what happened as they changed the code. Next time I get sent work experience pupils, I will definitely use the cyber-dojo again, and hope it might inspire a young person to learn how to program.

References

[1] https://dl.dropboxusercontent.com/u/77626588/infoForResearchParticipantsA4.pdf

[2] http://cyber-dojo.com/

Have you experienced something which has changed your perspective, had a positive effect on you, or just given you a buzz? Let us know at cvu@accu.org.

Write for us!

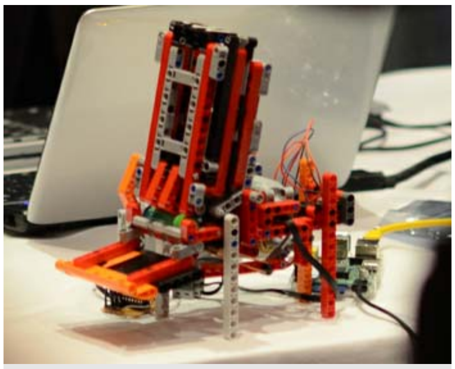
C Vu and Overload rely on article contributions from members. That's you! Without articles there are no magazines. We need articles at all levels of software development experience; you don't have to write about rocket science or brain surgery.

What do you have to contribute?

- What are you doing right now?
- What technology are you using?
- What did you just explain to someone?
- What techniques and idioms are you using?

For further information, contact the editors: cvu@accu.org or overload@accu.org

ACCU Conference 2013 (continued)


Sometimes it's a hardware problem!
(Photo courtesy of Stephan Eggermont)

Riverblade, and some of the lessons we've learnt along the way.

The title of the talk is a reference to the fact that when we first started in 2004, we seemed to be the only tech startup on the networking circuit in Bournemouth (but there were however a lot of hairdressers and garden designers...).

We thoroughly enjoyed the conference, so if you've not been to an ACCU Conference before we can certainly recommend it! Copies of the slides for most of the sessions which took place during the week are available via the ACCU website.

Finally, if you want to get a good feel for the atmosphere at the conference, you can do a lot worse than to check out the photos, videos, tweets and slides on the ACCU 2013 Eventifier page.

References

[1] http://agileonthebeach.com/

[2] Barrett, Louise (2011) *Beyond the brain: how body and environment shape animal and human minds* Princeton University Press.

Acknowledgements and copyright

Unless otherwise specified, all photos are copyright Anna-Jayne Metcalfe and are reproduced with her permission.

Jon Jagger and Kevlin Henney closing ACCU 2013

Bookcase

The latest roundup of book reviews.

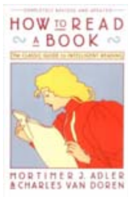
If you want to review a book, your first port of call should be the members section of the ACCU website, which contains a list of all of the books currently available. If there is something that you want to review, but can't find on there, just ask. It is possible that we can get hold of it.

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.

Jez Higgins (jez@jezuk.co.uk)

How to read a book : The classic guide to intelligent reading



By Mortimer J. Adler & Charles Van Doren. ISBN-13: 978-0-671-21209-4

Reviewed by Ian Bruntlett

First an acknowledgement, On 1st April, on the accu-general mailing list, Huw Lloyd recommended this book. I ordered it and commenced devouring it armed with note paper, a pencil and a highlighter pen. This book is in four parts and I will deal with them in turn.

Part 1, 'The dimensions of reading' discusses the basics of reading – why do we do it? And 'How do we do it?'. There are different stages of reading, the first being Elemental Reading. The second, (the two types of) Inspectional Reading. The first type is systematic reading, the second type is superficial reading. The general idea is that we read different genres of books differently and the level of concentration and study suitable for one work will be inappropriate for another. The final chapter, 'How to be a demanding reader' introduces the Essence of Active Reading as four basic questions that a reader should ask (in differing forms) about a book being read. They are: 1) What is the book about as a whole? 2) What is being said in detail, and how? 3) Is the book true in whole or part? And 4) What of it? (is the book significant?).

Part 2 is 'The third level of reading: Analytical reading'. The first rule is that you must know what kind of book you are reading and to adapt your reading style accordingly. The second rule is that you must be able to summarise the book in a couple of sentences. Rule three is more demanding of the reader and the book – identify the major parts of the book, how they interact with one another and how they, together, create a greater whole. Rule 6 is introduced here – 'Mark the most important sentences in a book and discover the propositions they contain'. It mentions Intrinsic Reading (reading a book quite apart from other books) and Extrinsic Reading (reading a book as part of a collection of related books).

Part 3, 'Approaches to Different Kinds of Reading Matter' was extremely rewarding to read. To summarise, it covers, practical books, imaginative literature, stories, plays and poems, History, Science and Mathematics, Philosophy and Social Science.

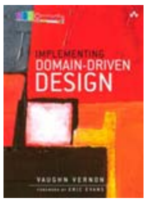
Part 4 is 'The ultimate goals of reading'. This seems a strange title to me. However, the contents are still interesting. It introduces the 4th level of Reading – Syntopical reading which is an approach to take when reading different books on the same topic.

Appendix A provides the reader with a recommended reading list. Appendix B Exercises and tests can be used by the reader to see if they have grasped the concepts in this book.

If this review hasn't answered the question 'Shall I read this book?', look up 'How to read a book' on Wikipedia.

The bulk of the book (excluding appendices and index) consumes 346 pages. If you have time to read it and you intend to read broadly and not just within your own field, I recommend this book.

Implementing Domain-Driven Design



By Vaughn Vernon, published by Addison-Wesley.

Reviewed by Ian Bruntlett,

Highly recommended.

This book builds on the prior work, *Domain-Driven Design : Tackling the Complexity in the Heart of Software* (Eric Evans,2004). Eric Evans introduced the idea of 'Ubiquitous Language'. The 'Ubiquitous Language' is something that different stakeholders in a development project can communally agree on.

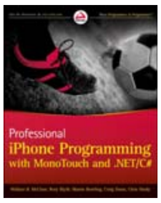
This book's examples are samples written in Java or C#. There is plenty of discussion about design or implementation techniques, especially leaning upon the samples based on a Java based multi-threading, multi-user, networked system. It discusses different strategies, including their relative strengths and weaknesses.

When discussing the DDD, the role of a Ubiquitous Language is introduced. A business is defined by its purpose – this becomes its core domain. Lesser domains are 1) supporting sub-domains (business specific things) and 2) generic sub-domains (things that can be bought in or provided by other development teams. It discusses different architectures – I must confess I hadn't heard of the Hexagon architecture (aka Ports and Adapters) and really like it.

From a personal point of view, I particularly really like these concepts introduced by this book. A lot of the book makes sense from some C++ Builder/SQL systems I once developed. I particularly like the way the book's concepts can help you avoid building systems that are a big ball of mud or a data silo.

This is an excellent book if you need its advice. It could do with a glossary though – I improvised by referencing Wikipedia for unfamiliar terms. It does have a useful bibliography with references to many papers on the Internet.

Professional iPhone programming with MonoTouch and .NET/C#



By McClure, Bowling, Dunn, Hardy and Blyth Wrox - ISBN 978-0-470-63782-1

Reviewed by Paul Johnson

Highly recommended

I will start this review by coming clean. I hate Obj-C. Always have. I remember trying it years ago and thinking that it was just an unholy mess that deserves to be consigned to history as a bad mistake. For some reason, Apple decided to use it for iPhone/iPad development and, well you know what happened next.

Thankfully, with the advent of Xamarin.iOS (the new name for monotouch) apps can be written quickly and easily in .NET. The compiler then does some clever stuff and the resulting code is allowed to be distributed on the Apple store. All I can say is thank goodness!

More over, thank goodness for this book! As with the Android with Mono for Android book, there are precious few books for .NET developer writing code on Apple devices. Thankfully, this book does a damned good job at filling that void.

Writing for Apple devices is a much tougher act than writing for Android and the first couple of chapters cover how to create the UI using Xcode and how to connect the buttons up to the viewer so that code can be written. As with the Android book, the authors go to great lengths to simplify this process and to keep everything as clear as possible.

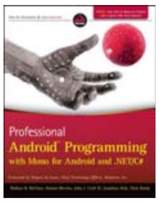
Everything up to chapter 5 is to do with the user interface. You may think 4 chapters is a fair bit for UI, but given that Apple places such a high value on the user experience, this can be considered slightly short.

The next part of the book is more the real nuts and bolts of iPhone programming – data, more on the UI, maps and all of the funky parts you expect on the phone including (importantly) how to communicate with the outside world, video, sound and talking to other applications.

Unlike the Android book, there is a chapter given over to knowing just enough Obj-C to get away with things. Why is this? Well, despite Xamarin.iOS doing a brilliant job, the likes of NSObject, NSString et al are still there and it is useful for bringing over code in ObjC to C# where the book has not mentioned something (come on, it's a book and has a finite number of pages and despite their best efforts, not every problem that can be encountered has been covered!)

Quite simply put, for those writing code on Apple devices using Xamarin.iOS, this is a must.

Professional Android Programming with Mono for Android and .NET/C#



By McClure, Blevins, Croft, Dick and Hardy Wrox - ISBN 978-1118102275

Reviewed by Paul Johnson

Highly recommended

Xamarin.Android (as it is now known) suffers from one simple thing – a lack of any sort of reference material. There are scant few books covering .NET programming on Android devices and programming with .NET on Android is not the same as programming for Android in Java. For a start, it's easier!

If you know .NET already, why go through the curve of learning Java?

What this book provides the user with are simple to follow examples of not how to code in C# (that is wisely left to other books), but how to code for Android using C# and to be able to access video, sound, maps and pretty much any other aspect of using an Android device. It is an invaluable addition to any book shelf. Everything in the book is clear and well explained and what is even more important, I was unable to find any of the examples that didn't just work straight off or leave me in any confusion on how to code for Android using C#.

The book though does have two omissions, one of which is perhaps excusable as it can be argued as being outside of the scope of the book and that is helpful advice in porting native Java source to C#. Bringing code from the plentiful Java examples to C# is of immense use. The other omission is memory management.

Xamarin.Android suffers from a memory problem. Say you create a **`List<T>`** in .NET for the Android device. When the activity goes out of scope, the GC does its job and cleans the **`List<T>`** up – or so you think. The GC cleans the .NET pointer to the underpinning Java **`List<T>`**, but not the underpinning Java List. That has to be explicitly removed. Another memory issue is in handling bitmaps and bitmap manipulation. These can be massive memory hogs and in some cases, cause the app to crash horribly.

Despite these, this book is just what the doctor asked for. A simple to follow and clearly written text that shortens the learning curve.

I hope there is a second edition which corrects these omissions and adds in animation as well as bringing it up to date with Android 4.2 being covered.

Nginx HTTP Server

By Clement Nedelcu, published by Packt Publishing ISBN: 978-1849510868

Reviewed by Alan Lenton

Nginx (pronounced as 'Engine X') is a lean, mean and fast web server. It's open source, and designed to serve pages fast. We use it as work, and, while it is not as well known as Apache, and maybe not as comprehensive, you don't need a Ph.D. in chaos theory to understand and write its configuration files!

This book is an excellent, and thorough, introduction to how to set up and use the server. Nginx is a modular server and the core modules, together with the rewrite module, the server-side include module, and the SSL module are covered in sufficient depth that anyone with a reasonable level of sysadmin knowledge would be able to set up the modules properly and safely. Other 'standard' modules are covered briefly, but third party modules are not covered at all. At first I thought that was an unfortunate omission, but on reflection, given the speed with which third party modules are developing and changing, that was probably a wise decision.

Once the author has covered all the basics there are a number of interesting and useful chapters covering other related topics. One of them covers using Fast-CGI both with Python and PHP. This is excellent, and includes a basic explanation of what CGI is and how to interface and use Nginx with the PHP-FPM and python based Django frameworks. Another chapter teaches you how to use Nginx as a reverse proxy along with Apache, and a third chapter covers the tricky business of moving your web site from Apache to Nginx.

The only weird thing about this book is the first chapter, which appears to be a potted newbie's guide to Linux system administration. I've no idea why it's there, perhaps the author's contract with the publisher specified that the book had to be over 300 pages long? Most people trying to set up a web server will probably know at least some system administration. If that's the case, my advice is to start at chapter two.

I was impressed by this book (actually I was also very impressed with Nginx) and I would definitely recommend it to anyone coming to Nginx for the first time.

ACCU Information
Membership news and committee reports

accu

View from the Chair

Alan Griffiths
chair@accu.org

Since my last report the main development has been that the Membership Secretary elected by the AGM (Craig Henderson) unexpectedly resigned. The committee has co-opted last year's Membership Secretary (Mick Brooks) to replace him until a more permanent solution is found. We can all thank Mick for stepping into the job he so recently vacated – but we do need a volunteer to take over the post as Mick will not be continuing next year.

The following are the responsibilities of the membership secretary:

- Consult website once per month to estimate number of journals required at next printing, and notify the production editor.
- Consult website once per month to retrieve mailing info for the journal, and forward to the distributor.
- Receive excess journals each month, and store up to a year's worth.
- Consult bank statement once per month and enter standing order payments into the website admin interface.
- Chase any standing order underpayments.
- Send journals out to members who have missed an issue.
- Send journals out to prospective members, groups or conferences that request them.
- Keep an eye on the stream of automated renewal and joining payments.
- Answer enquiries to accumembership@accu.org, typically:
  - prepare invoices and receipts
  - resolve payment troubles
  - handle address changes
  - advise on the benefits of membership

Keep some stats on the size and makeup of the membership, reporting back to the association.

Prepare reports for the treasurer on the donations that members make when they join or renew.

If anyone is interested in the role, please contact me (or Mick) – the committee would like to co-opt any potential candidates so that they can understudy Mick during the remainder of the year (while he is available to help) and will have a feel for the job before the AGM.

Distributed committee meetings are becoming increasingly routine: Of the nine members attending the last meeting there were three countries represented, and there were only three physically present. In the past the need to attend committee meetings in the UK has been a barrier to members from other countries participating.

We've not yet resolved the problems with the failure to have accounts ready for the AGM. As The Treasurer was unable to attend the last committee meeting we deferred discussion to the next one.

Other aspects of the ACCU are going smoothly, the journals have a new distributor. The mailing lists are busy and the website is being updated to show what is happening in the organisation.

We still have no volunteer to moderate the accu-contacts mailing list. This isn't an onerous task (there are a few emails each week to classify as “OK”, “Spam” or “needs fixing”) but we don't currently have a replacement for this role. Is this something you could do for the ACCU?