the magazine of the accu

www.accu.org

{cvu}

Volume 28 • Issue 2 • May 2016 • £3

{cvu}

Volume 28 Issue 2
May 2016
ISSN 1354-3164
www.accu.org

Editor
Steve Love
cvu@accu.org

Contributors
Baron M, Pete Goodliffe, Alan Griffiths, Kevin Highley, Roger Orr, Mrs Trellis

ACCU Chair
chair@accu.org

ACCU Secretary
secretary@accu.org

ACCU Membership
Matthew Jones
accumembership@accu.org

ACCU Treasurer
R G Pauer
treasurer@accu.org

Advertising
Seb Rose
ads@accu.org

Cover Art
Pete Goodliffe

Print and Distribution
Parchment (Oxford) Ltd

Design
Pete Goodliffe

accu

More Than One

As I write this, I've not long returned home from the ACCU Conference. If you were there, I hope you had a productive and fun time of it. As ever, it's been a full-steam-ahead schedule of fascinating talks, interesting people and insufficient sleep, not to mention the annual agonising over which talks to miss and which to attend.

There is often a recurring theme at the conference, and this year was no exception. I don't think anyone who attended would disagree that the theme was 'diversity'. The human diversity issues raised, however, are mirrored in technological diversity: to accept and embrace many *different* forms. The ACCU conference is often a rich environment for raising awareness of new and exciting technologies, finding new and exciting ways of using existing technologies and attempting to shine a light on the modern with insights from classic techniques.

There were talks specifically about using multiple languages to achieve the same, or similar, goals. It's a fairly well-known tip from The Pragmatic Programmers and others that learning new programming languages informs our understanding of the languages we already know, whilst simultaneously broadening our perspectives on different styles, techniques and technologies. This is valuable because it can help prevent us from becoming stagnant in our views and philosophy.

I'm aware that I risk extending the metaphor too far by making the comparison with human diversity and acceptance; however, there are parallels that are worth considering. We pride ourselves as technologists in welcoming and accepting technological change for the positives it brings to us. We should not ignore human diversity for much the same reasons – it enriches our world and our outlook.

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

REGULARS

FEATURES

SUBMISSION DATES

C Vu 28.3	1st June 2016
C Vu 28.4:	1st August 2016
Overload 134:	1st July 2016
Overload 135:	1st September 2016

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

Encryption

Kevin Highley implements a common technique for secure communication.

Keeping secrets has been important for thousands of years. Modern communication systems and computers have given us new ways to store and exchange information, and new ways for the wrong people to read it. Encryption is a tool to tackle such problems, but what is it, and as a programmer, how do I do it?

Conventional encryption is about securing communications. A message is converted to a form that only the intended recipient can read. Often this depends on information that only the sender and recipient know, used as a key to 'unlock' the message.

It is assumed that any intercepting party knows about the encrypting methods. Obviously, if an enemy doesn't know a message exists they won't be looking for it. Even if they catch a message, we hope they won't recognize it as such, or know how to read it. The only safe assumption, however, is that the enemy knows everything they could possibly know. Security through obscurity sometimes works, but you never know if someone even more obscure is watching you.

Simple systems

A message is a series of symbols, for our purposes a string of characters. Their meaning can be disguised by re-arranging their order and/or by substituting for individual characters with other characters.

For example, let us take the message "What shall we do tonight?" A simple re-ordering is achieved by arranging the letters in an array horizontally, and reading them out vertically. 25 characters, so a 5x5 array.

```
W h a t  
s h a l l
  w e   d
o   t o n
i g h t ?
```

Coded, the message becomes "Ws oihhw gaaethtl ot ldn?"

Replacing each character with a letter from a certain distance further along in the alphabet (or for us, the ascii code) gives a code reputedly used by Julius Caesar. If the offset is +1:

"What shall we do tonight?"

becomes

"Xibu!tibmm!xf!ep!upojhiu@"

Both types were used in ancient times. The Greeks supposedly spirally wrapped a strip of leather round a stick, and wrote along the stick. The message could only be read if wrapped around the same diameter stick. This gives a version of the re-ordering code above. The stick diameter key corresponding to the dimensions of the array. In Julius Caesar's code, the key is how many letters to shift along.

These approaches yield exciting looking messages, but would be easy to unravel.

I can see that '!' Is probably space and '@' a punctuation mark. On a longer message, and with some idea about what the message might be about, and what language it is in analysing letter frequencies will soon reveal all. A more elaborate character substitution system than to just shift *n* characters along would not help. The commonest character in the output still represents the commonest character in the input.

We can make the code look much more exotic by using strange fonts, Egyptian hieroglyphs or made up special symbols. So long as each letter in our plain text is always represented by the same symbol it is relatively simple to break the code. In normal English text ' ' and 'e' occur much more often than 'x' and 'q'. Count the number of times each symbol in the encoded text appears, and replace each symbol with the letter that occurs with the most similar frequency in English (or whatever language you think the message is in). This will often immediately give you something that is almost readable, needing only a few letter allocations to be swapped about. This gets easier with larger messages, and messages in which you can guess that certain words or people's names are likely.

A less obviously structured re-ordering of the characters would be better, and we need a substitution system in which the same character might be represented by different characters each time it appears.

Computers and keys

Those older systems had keys of only a few letters or digits, there are only 26 letters in the alphabet, and only a limited number of practical stick sizes. In current terms they are 4 or 5 bit codes, they have only 2^4 or 2^5 possible keys. Such small key encryption systems can be broken by a sufficiently powerful modern computer. Simply run a version of the decrypting algorithm with every possible key. We can devise systems with more complex keys, but while a human codebreaker might be daunted by a system that required thousands of tries to find the right key, computers thrive on big numbers. WWII code crackers would have been delighted with the power of a modern laptop, but would have reminded you: if you've got this, what have the big boys got? A coding system that would take, even with the world's top computers, years to break seems secure enough, very little intelligence stays useful for more than months. However – are you sure you know just how fast those top machines are? Are you sure there is no easier back-door way to unpick your encryption, which could improve decryption rates by orders of magnitude?

How big does a key need to be? Current super computers are chasing 100 petaflops (10^{17}) and there are about 10^8 seconds in three years. A reasonably lucky code breaker might find the right value after trying 10% of all the possible values. Perhaps a secret, clever, dedicated code breaking system can test a key value in the time the machines we *do* know about take to do a floating point operation (note how fast colossus was compared with general purpose machines of the late 40s). We need keys giving more than 10^{26} possibilities, more still if we suspect there may be any cracks in the encryption method. $10^{26} \sim= 2^{87}$. Even with the more reasonable assumptions, that it might take 1000 operations to test a key, and that results are needed within a day, we still need a key with $> 10^{20}$ possibilities, that's about 2^{67}. The argument about 64 bit versus 128 bit encryption becomes clear. Even a perfect code system with no flaws and back doors might be breakable by brute force if it uses 64 bit or smaller keys. For the brute force decoder, as the key size goes up the problem becomes selecting probable messages (e.g. ones with recognizable words in them) from the vast number of possibles generated.

A large key makes brute force (try everything) approaches expensive in computer time and thus impractical. We need an algorithm that makes it

KEVIN HIGHLIEY

Kevin Highley started programming at IBM in the early 1970s. He has degrees in Maths and Music, likes programming in assembler, C and C++, and plays the bass clarinet, and the concertina. He can be contacted at tokamak@outlook.com

hard work to test each key. We don't want the breaker to be able to reject a key after just a few characters, we want no output until they have decoded most of the message. We also need to avoid anything that lets them home in on a solution. Trying to find our way to a particular level on a smooth slope is much easier than trying to find a stone of a particular weight from a pile of similar stones by weighing one at a time. If we were trying to find the zeroes of a function it is much easier if the function has smooth derivatives, and much harder if it seems to be a random step function.

One time pads

One way of defeating the frequency analysis attack is to use Julius Caesar's substitution method, but have a different offset for each character in the message. The key is now a list of numbers the same size as the message, each number indicating the amount to shift the corresponding character in the message. The large key size is a problem: it is too big for people to remember, and if written down and passed around, it is possible for an enemy to learn it. Another weakness is that the same pattern is used for every message. If many messages have the same format (e.g. starting with the sender's name and the date) the same pattern will appear at the start of each coded message, providing something for the code breaker to attack. Adding a re-ordering step makes patterns harder to spot, but not impossible. A solution is to change the list of offset numbers frequently, although this leads to problems of how you distribute the large and numerous keys. If a new table of numbers is used for every message this becomes a 'one time pad' system, somewhat cumbersome, but completely secure so long as the particular table used has no pattern and stays secret.

The one time pad system was widely used, despite the problems of key distribution. Many tables of numbers were prepared, all different and serial numbered. A field agent would be issued with some. When contacting base they would use a table to encode their message, and then destroy the sheet containing that table. The only copy of that particular table, identified by it's serial number, now exists at base, so only they can decode the message.

The Enigma

Another way of changing the coding for each character was enigma type machines. They had a series of wheels, each of which generated a substitute for whichever letter you fed to it. Several wheels in series providing a more elaborate pathway. Pressing a key on the machine caused a letter to light up on the output. The clever bit was that the wheels were moved on after each letter had been pressed, generating a new sequence of connections, and a new coding for the next letter. The key for this system involved knowing the wiring patterns of all the wheels, which wheel was in each slot in the machine, and where each wheel has been rotated to at the start of the message. There were also jumpers that controlled other variables. All in all a fiendishly complicated device. The story of how these codes were broken at Bletchley Park is well worth a read and visit if you don't already know it.

Other methods

There are other ways of sending secret messages. Some are variations on the substitution and re-arrangement types. Public key encryption is a system where the key used to encrypt is different to the one needed to decrypt. It is important for much of web security. As far as I know it depends upon being able to find the prime factors of very large numbers, which is a difficult problem on it's own. Code systems where symbols stand for concepts rather than characters are hard to break. They overlap with the problems of trying to understand ancient languages. The code talkers of WW2 and the book code, as used in a Sherlock Holmes story, did not need a computer, nor do slang, jargon and the ever evolving languages of youth, designed to keep parents in the dark. These I can ignore in the next section, on using a computer for encryption. As for quantum methods, well, I just don't know how they work.

Book code: Message is sent as a series of number triads, each representing a word from a book selected by page:line:word. If needed, number quads can represent individual letters. The key here is the book to be used.

Code talkers: Messages were sent as conversations between members of a native American tribe in their own language. There were only a few speakers of the language left, and it was not available in a written form, so there was little chance of an enemy understanding it.

To be pedantic most of what we have been looking at should be referred to as ciphers rather than codes. Ciphers tend to have a 1:1 relationship between plaintext character counts and disguised text counts, codes can have complex ideas represented by short sets of symbols.

Programmers do it with a computer

Fortunately for the code writer (if not for big brother) it is simple to implement, scrambling, one time pad, and enigma type codes on almost any computer.

My programs frequently do the same thing to all the elements of an array. All the arrays are the same size, 256 bytes. To save writing the same code repeatedly:

```
#define Ri for(i=0;i<=255;i++)
```

Table (aka wheel) of size 256, and contains one each of each possible value. Why?

I am assuming the enemy knows everything, so I might as well use sizes that are convenient for computing. In assembler, I used 8 bit characters, pointers, and arithmetic. 256 eliminates the need for range checking, an 8 bit pointer can't get out of a 256 array. All different values reduces the key size from 256^{256} ~= 10^{616} to a mere 256! ~= 10^{506} – still a big number. By keeping all the numbers and merely re-arranging each pass I hope to avoid a wheel ever collapsing to all zeros, or other problem pattern. An all different 256 array can be used as a scrambling tool as well as a one time pad tool.

Characters are often stored on computer as 8 bit values. Those 8 bit values could also be thought of as numbers in the range 0–255. Using 8 bit unsigned arithmetic, if I add numbers in the range 0–255 to the numbers representing each character in my message I will get a new collection of numbers in the range 0–255. With 8 bit arithmetic, overflows are discarded.

If the plain text is in an array `uint8_t a[256]`
and our random numbers key in `uint8_t x[256]`
then `Ri a[i] += x[i];`
provides the encryption step, `a[]` now contains the encrypted message. `a[]` and `x[]` need to be unsigned `char` or `uint8_t` to get the right overflow behaviour. That's it! unbreakable encryption in one statement.

Part of the security problem is to ensure that the machines you are using are not already infested with some key grabber or screen shot program that records the plain text before you encrypt it. This is easier to check with less software layers between you and the hardware. The code examples were first imagined in assembler, to run on very minimal machines. This was changed because A, My friends no longer use such machines, and B, few find assembler easy to read, and the point of writing the code was to illustrate an idea. The current program in minimal C running on the console (`stdin`/`stdout`) should run on many machines. I want you to modify and compile for yourself, at this level of security code, don't trust anything where you haven't seen and understood the source. For better security... am I paranoid, or is that Windows 10 watching me? Run your encrypt and decrypt programs on a small machine that is not connected to any network or other machines, and transfer the encrypted file via a memory stick to the machine that sends it.

If I later subtract the same numbers, I will get back my original character values, underflows are also discarded. If all the numbers added had been the same we would have had Julius Caesar's code, but as the numbers added are all different, and nearly random (no patterns to find and exploit), we have a strong code. The key is now a table of nearly random numbers the same size as the message.

Listing 1

```
uint8_t a,b,z;
int i;

Ri x[i] = i; // Fill x array with values 0 - 255
for (i=0; i<2000; i++) {  // Swap random pairs
                          // of values 2000 times
  a = rand(); b = rand(); // to give all possible
  z = x[a];     // 8 bit numbers, but in a random
  x[a] = x[b]; // order. Like Eric Morecombe's
  x[b] = z;     // music "All the right notes, but
}               // not necessarily in the right
                // order."
```

Decrypt is just `Ri a[i] -= x[i];`

The random number table approach is good, but the table must change frequently. If we make our table of random numbers completely new for each message we have a 'one time pad' system. If we were to move to a different place in the table after each character we have an enigma-like system.

We need random number tables. Assuming the random generator has been seeded, the code in Listing 1 generates one.

> It might be better to restrict `rand()` to produce only numbers in the range 0–255, here it seems to work, presumably by taking only the bottom 8 bits. There are probably many better ways, but it is important to be your own carpenter when making random number tables. The only way to read a one time pad system is to know the tables, if you copy some else's tables or their generating code that might just be discoverable. It is also unwise to rely on the system supplied random number generator, some are flawed. Using the random number indirectly as here may insulate you from some generator foibles.

With this sort of random number table containing one each of the numbers 0–255 an order scrambling routine is very simple to implement.

With `x[]` the random table, `o[]` the plain text, and `a[]` the scrambled text:

To scramble,	`Ri a[i] = o[x[i]];`
and to unscramble again,	`Ri o[x[i]] = a[i];`

The tools we have so far would enable us to produce a secure email encrypting program. Generate a few million random number tables, store them on your machine, copy onto a flash drive or DVD and give it to the intended recipient. Convert your message to an 8 bit unsigned character string, add a random number table, then use the same table to scramble. Send the encrypted array, and tell recipient which random table to use to decrypt it.

> Potential problem?
>
> If we re use the random numbers, even if changed daily, a repeatedly sent message that was all one character, such as blank, would be converted to an offset version of the table. That may not seem to be a problem, the enemy doesn't know that you have sent an image of today's pattern. But, imagine something resting on a keyboard generating twenty pages of white space, even a dumb cracker might notice twenty consecutive identical messages, and figure out what had happened. Such mishaps have been the clue that cracks a code.

So far so good, but the receiver may fall into the wrong hands, and the enemy obtain the random number flash drive, and details of your encrypt/decrypt algorithm.

We can dispense with the flash drive by making our machines scramble the random number array after each message. We need to do exactly the same scrambling at both transmitter and receiver. This has the advantage that even if a machine is captured it can only decode the next message, the tables used for previous messages are gone. The one thing that transmitter and receiver have in common is the message, so use that to scramble the table.

Random table in `x[]` encrypted message in `a[]` temporary `z`, all `uint8_t` (see Listing 2). The same code is used at both ends after encrypting or decrypting a message.

Listing 2

```
Ri {
  z = x[i];
  x[i] = x[a[i]];
  x[a[i]] = z;
}
```

The system now changes the random table every message. A single stage provides fairly solid encryption, the system is behaving almost as a one time pad. An attacker with access to the receiver, all the encoded and some decoded messages would still be able to work out the current settings, and, knowing the scramble algorithm, 'unzip' the whole chain of messages. The next stage makes that more difficult.

It would be useful to re-introduce small keys, something the user can remember, which stops an enemy immediately using a captured machine. We also need to make the decrypt more clock cycle hungry so that a captured machine cannot be quickly put back into action with brute force discovered small keys. To this end I now introduce multiple random number tables, and a more complicated encryption algorithm. Our small keys select the order in which the tables are used, and where in each table we start.

The main encryption stage uses ideas from the enigma code machine (that's why I start calling random number tables wheels). As a character is encoded by a wheel it also moves that wheel to a new position. The process is repeated 5 times with five different wheels.

We now have 6 wheels, they are implemented as the arrays `x[256][6]`, and 6 keys, stored in array `k[6]`, each holding a value 0–255. The array `w[6]` represents the slots in our imaginary enigma machine, so `w[n]` contains the number of the wheel in slot `n`. `k[n]` contains the starting position for the wheel in slot n. Slot 0 has already been used with the initial encode and scramble. We count along the slots with `j` and have 6 working arrays `a[~][6]`. The already partly encrypted message starts in `a[~][0]`, and ends up in `a[~][5]`.

```
for(j=1; j<=5; j++) {   // Main enigma loop
  k = key[j];           // Wheel to start position
  Ri {
    a[i][j] =           // Value from wheel added
      a[i][j-1] + x[k][w[j]];
    k = [a[i][j-1]][w[j]]; // Wheel moved to
  }                        // new position
}             // according to character just read
```

Finally the code wheels are scrambled using the intermediate stages of the above encryption, (which being ephemeral are not available to a code breaker), to control the scramble. Temporary swap value `z` is again `uint8_t`.

```
for(j=0; j<=5; j++) {
  Ri {
    z = x[i][w[j]];
    x[i][w[j]] = x[a[i][j]][w[j]];
    x[a[i][j]][w[j]] = z;
  }
}
```

The main encrypt loop and the code wheel scrambling are separate here to make it easier to see what is happening. They could be combined into one loop to reduce memory use.

The decrypt routines are almost a mirror image of the encrypt.

> I notice that the encoded message looks very different to the plain text – lots of unprintable characters. If messages are mostly printable characters then perhaps we should convert the encoded text to mainly printables. We will have to do that any way in order to use some communication channels. Or perhaps a first pass which assigns several of the 0–255 symbols to 'e' and 'space' and fewer to 'q' to make messages look more like noise across the whole 0-255. Anything which gives a breaker program less of a entry point is to be encouraged.

The demonstration program, available from https://github.com/KevinHighley/CryptoEntanglement, is deliberately minimal, I want it to run on almost any machine with a C compiler. Because I rely on 8 bit

Come Code With Me

Alan Griffiths outlines an Open Source project and invites contributions.

One of the advantages of working on Open Source projects is that is possible to talk about them and the code involved with few restrictions. In this case I'm working on a project that would greatly benefit from wider input and I would especially welcome contributions from ACCU members.

This is a C++ 14 project, with a lot of potential features that are not important to my immediate goals, so I'm unlikely to get to them any time soon. There's a range of work available varying from entry level programmer to domain expert, so if people want to use it as a context for trying C++ 14 features (or even C++17 features) that's great.

I've not heard anything about ACCU 'mentored' projects recently, but if any ACCU members want to use this project as a basis for this, then I'd be very happy. And I'm willing to provide support for this.

The project is about developing a 'Desktop Environment' or 'Shell'. There are a lot of these in the Linux world; a few examples (in no particular order): KDE, Gnome, Unity7, Cinnamon, LXDE, and Awesome. Canonical's interest lies in Unity8 – which is the 'convergent' shell currently being used on phones and tablets and in preparation for desktop use.

Introducing the Mir Abstraction Layer

The principle Open Source project I've been working on for the last few years is Mir. Mir is a library for writing Linux display servers and shells that are independent of the underlying graphics stack. It fits into a similar role as an X server or Weston (a Wayland server) but was initially motivated by Canonical's vision of 'convergent' computing.

The Mir project has had some success in meeting Canonical's immediate needs – it is running in the Ubuntu Touch phones and tablets, and as an experimental option for running the Unity8 shell on the desktop. But because of the concentration of effort on delivering the features needed for this internal use, it hasn't really addressed the needs of potential users outside of Canonical.

Mir provides two APIs for users: the 'client' API is for applications that run on Mir and that is largely used by toolkits. There is support for Mir in the GTK and Qt toolkits, and in SDL. This works pretty well and the Mir client API has remained backwards compatible for a couple of years and can do so for the foreseeable future.

The problem is that the server-side ABI compatibility is broken by almost every release of Mir. This isn't a big problem for Canonical, as the API is fairly stable and both Mir and Unity8 are in rapid development: rebuilding Unity8 every time Mir is released is a small overhead. But for independent developers the lack of a stable ABI is problematic as they cannot easily synchronize their releases to updates of Mir.

My answer to this is to provide a stable 'abstraction layer' written over the top of the current Mir server API that will provide a stable ABI. There are a number of other goals that can be addressed at the same time:

- The API can be considerably narrowed as a lot of things can be customized that are of no interest to shell development;
- A more declarative design style can be followed than the implementation focused approach that the Mir server API follows; and,
- Common facilities can be provided that don't belong in the Mir libraries.

At the time of writing the Mir Abstraction Layer (miral) is both a proof-of-concept and a work-in-progress but may be of interest as a modern C++ codebase to experiment with.

Building and using MirAL

These instructions assume that you're using Ubuntu 16.04LTS; I've not tried earlier Ubuntu versions or other distributions.

You'll need a few development and utility packages installed, along with the mir development packages (if you're working on a phone or tablet use mir-graphics-drivers-android in place of mir-graphics-drivers-desktop):

ALAN GRIFFITHS

Alan Griffiths has delivered working software and development processes to a range of organizations, written for a number of magazines, spoken at several conferences, and made many friends. He can be contacted at alan@octopull.co.uk

Encryption (continued)

unsigned arithmetic overflow behaviour, be careful if you change anything, not to mix **`uint8_t`** values with **`int`** counters. The 8 bit variables get promoted to 16 or 32, and 'interesting' things happen. The demo uses two sets of **`x[]`** arrays, **`xs[]`** are the arrays on the source machine, **`xd[]`** on the destination. I have a more elaborate version of the demo written using Qt widgets and C++. It works on my Linux mint setup, but I have not yet worked out how to port it to Apple, Windows and Android, which is supposed to be straight forward with Qt. This will probably have been done by the time this sees print – although I am being distracted by the classic travelling salesman problem at the moment.

Conclusion

Secure communication between devices can be achieved by the one time pad system. In the past this was restricted by the need to distribute coding pads with suitable sets of random number tables, more data than all the messages to be sent put together. The current price of memory sticks makes 16GB of random number tables generated, and physically passed between stations practical. No wonder the authorities say they only want to store information about messages, not content. The content of e-mails is potentially pretty secure anyway.

I suggest an alternative approach which doesn't need big random number tables. Set up a few random number tables, matched on transmitter and receiver, and mutate them as part of the process of sending and receiving each message keeping Tx and Rx in step. This also has the effect of making the receiving machine capable of decoding only the current message, even with the right user keys. A captured machine cannot decode past or future messages.

The old assumption was that the enemy has access to all message traffic, but not to either end. A safer assumption is that either end may be compromised at any time, and to restrict potential damage in that event as much as possible. ■

```
$ sudo apt-get install cmake g++ make bzr python-
imaging
$ sudo apt-get install mir-graphics-drivers-
desktop libmirserver-dev libmirclient-dev
```

With these installed you can checkout and build miral:

```
$ bzr branch lp:miral
$ mkdir miral/build
$ cd  miral/build
$ cmake ..
$ make
```

This creates `libmiral.so` in the `lib` directory and an example shell (miral-shell) in the `bin` directory. This can be run directly:

```
$ bin/miral-shell
```

With the default options this runs in a window on your X11 desktop (which is convenient for development). To run independently of X you need to grant access to the graphics hardware and specify a VT to run in. For example:

```
$ sudo bin/miral-shell --vt 4 --arw-file --file
$XDG_RUNTIME_DIR/mir_socket
```

The miral-shell example is simple, don't expect to see a sophisticated launcher by default. You can start mir apps from the command-line. For example:

```
$ bin/miral-run gnome-terminal
```

That's right, many standard GTK+ applications 'just work' on Mir (as GDK has a Mir backend). Not all 'GTK' applications work however: those that assume the existence of an X11 server will have problems.

miral-shell supports a lot of the familiar commands of a desktop environment. For example:

- Alt-Tab to switch applications;
- Alt-Grave to switch windows in an application;
- Alt-F4 to close;
- Alt+F11 to toggle fullscreen;
- Alt+left-button-drag (or three-finger-drag) to move; and,
- Alt+middle-button-drag (two-finger-drag) to resize.

To exit from miral-shell press Ctrl-Alt-BkSp.

Starting applications in miral-shell

If you have a terminal session running in the MirAL desktop (as described above) you can start programs from it. GTK, Qt and SDL applications will 'just work' provided that they don't bypass the toolkit and attempt to make X11 protocol calls that are not available.

```
$ gedit
$ 7kaa
```

From outside the MirAL session the 'miral-run' script sets a few environment variables to configure the Mir support in the various toolkits. (There's some special treatment for gnome-terminal as starting that can conflict with the desktop default.)

```
$ bin/miral-run gnome-calculator
$ bin/miral-run 7kaa
```

There are also some examples of native Mir client applications in the mir-demos package. These are typically basic graphics demos:

```
$ sudo apt-get install mir-demos
$ mir_demo_client_egltriangle
```

What does using the MirAL API look like?

The **`main()`** program from miral-shell looks like Listing 1.

The shell is providing **`CanonicalWindowManagerPolicy`**, **`TilingWindowManagerPolicy`**, **`spinner_splash`** and **`spinner_server_notification`**. The rest is from MirAL.

If you look for the corresponding code in `lp:qtmir` and `lp:mir` you'll find it less clear, more verbose and scattered over multiple files.

A **`WindowManagerPolicy`** needs to implement an interface for handling a set of events (see Listing 2).

The way these events are handled define the behaviour of the shell. This interface is going to change as part of my ABI stabilization work as it mentions some Mir server API types directly (for example, **`mir::scene::SurfaceCreationParameters`**) and an interface like this is a risk to long term ABI stability as adding new functions would break client code.

The principle interface for controlling Mir is similar (see Listing 3).

Again this still mentions a few Mir server API types and that needs fixing before miral is ready for release, but as this is implemented by the library (and not the client) there is less of an issue in using an interface for this purpose.

Exercises for the reader

As I said in the introduction, my focus is the MiraAL ABI and making it one that can maintain compatibility into the future. But there are also a lot of interesting possibilities for using that ABI and extending the miral-shell. I've tried to put some of these into a `tasks_for_the_interested_reader.md` file in the project (but I'm sure there are more).

It doesn't take long using miral-shell to realize that it is lacking such things as a 'launcher' and a 'status bar' and that the 'titlebars' are little more than a placeholder for the functionality that one would expect. Much of that will have no effect on the Miral API and I'm unlikely to get around to it. Other features (like animated transitions) will need additional support from the API and I'm hoping to get that right with the input of people trying to use it 'in anger'.

Canonical are committed to providing Mir on a range of platforms including phones, tablets and desktops. There is also work to provide it as a 'snap' in the 'internet of things'. The miral-shell already works in all

Listing 1

```
int main(int argc, char const* argv[])
{
  using namespace miral;
  return MirRunner{argc, argv}.run_with(
    {
      WindowManagerOptions
      {
        add_window_manager_policy<CanonicalWindowManagerPolicy>("canonical"),
        add_window_manager_policy<TilingWindowManagerPolicy>("tiling"),
      },
    display_configuration_options,
    QuitOnCtrlAltBkSp{},
    InternalClient{"Intro", spinner_splash, spinner_server_notification}
  });
}
```

Listing 2

```
class WindowManagementPolicy
{
public:
    virtual void handle_app_info_updated(mir::geometry::Rectangles const& displays) = 0;
    virtual void handle_displays_updated(mir::geometry::Rectangles const& displays) = 0;
    virtual auto handle_place_new_surface(
        ApplicationInfo const& app_info,
        mir::scene::SurfaceCreationParameters const& request_parameters)
    -> mir::scene::SurfaceCreationParameters = 0;
    virtual void handle_new_window(WindowInfo& window_info) = 0;
    virtual void handle_window_ready(WindowInfo& window_info) = 0;
    virtual void handle_modify_window(WindowInfo& window_info,
        mir::shell::SurfaceSpecification const& modifications) = 0;
    virtual void handle_delete_window(WindowInfo& window_info) = 0;
    virtual auto handle_set_state(WindowInfo& window_info, MirSurfaceState value)
    -> MirSurfaceState = 0;
    virtual void generate_decorations_for(WindowInfo& window_info) = 0;
    virtual bool handle_keyboard_event(MirKeyboardEvent const* event) = 0;
    virtual bool handle_touch_event(MirTouchEvent const* event) = 0;
    virtual bool handle_pointer_event(MirPointerEvent const* event) = 0;
    virtual void handle_raise_window(WindowInfo& window_info) = 0;
    virtual ~WindowManagementPolicy() = default;

    WindowManagementPolicy() = default;
    WindowManagementPolicy(WindowManagementPolicy const&) = delete;
    WindowManagementPolicy& operator=(WindowManagementPolicy const&) = delete;
};
```

Listing 3

```
class WindowManagerTools
{
public:
    virtual auto build_window(
        std::shared_ptr<mir::scene::Session> const& session,
        mir::scene::SurfaceCreationParameters const& parameters)
    -> WindowInfo& = 0;
    virtual auto count_applications() const -> unsigned int = 0;
    virtual void for_each_application(
        std::function<void(ApplicationInfo& info)> const& functor) = 0;
    virtual auto find_application(
        std::function<bool(ApplicationInfo const& info)> const& predicate)
    -> Application = 0;
    virtual auto info_for(std::weak_ptr<mir::scene::Session> const& session) const
    -> ApplicationInfo& = 0;
    virtual auto info_for(std::weak_ptr<mir::scene::Surface> const& surface) const
    -> WindowInfo& = 0;
    virtual auto info_for(Window const& window) const -> WindowInfo& = 0;
    virtual auto focused_application() const -> Application = 0;
    virtual auto focused_window() const -> Window = 0;
    virtual void focus_next_application() = 0;
    virtual void set_focus_to(Window const& window) = 0;
    virtual auto window_at(mir::geometry::Point cursor) const -> Window = 0;
    virtual auto active_display() -> mir::geometry::Rectangle const = 0;
    virtual void forget(Window const& window) = 0;
    virtual void raise_tree(Window const& root) = 0;
    virtual void size_to_output(mir::geometry::Rectangle& rect) = 0;
    virtual bool place_in_output(mir::graphics::DisplayConfigurationOutputId id,
                                 mir::geometry::Rectangle& rect) = 0;
    virtual ~WindowManagerTools() = default;
    WindowManagerTools() = default;
    WindowManagerTools(WindowManagerTools const&) = delete;
    WindowManagerTools& operator=(WindowManagerTools const&) = delete;
};
```

these environments (for a suitably forgiving value of ‘works’) and is an early opportunity to learn about this alternative to X11. ■

Acknowledgements

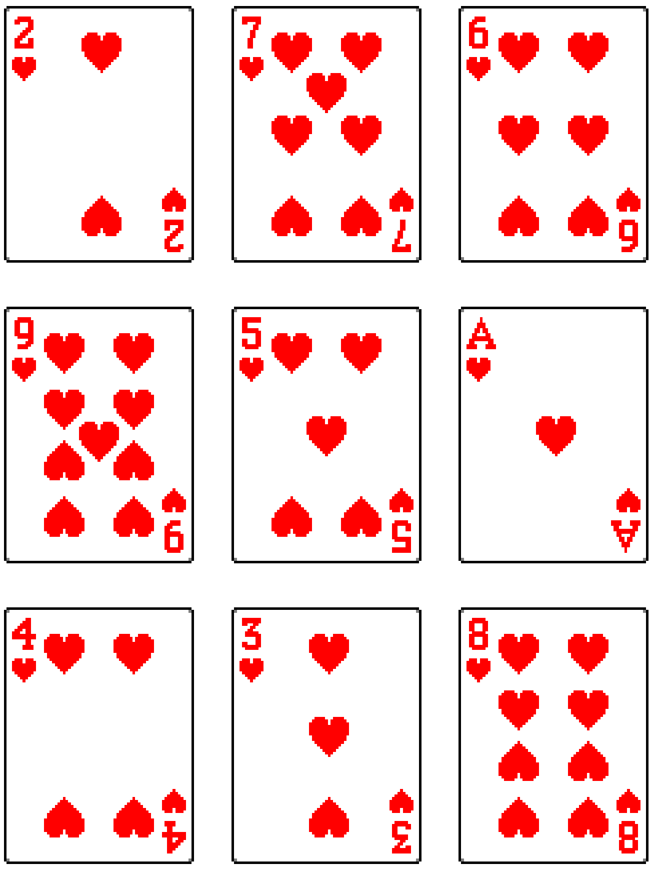
Thanks to Sam Spilsbury for his comments on an early draft and to Steve Love for dealing efficiently with a late submission.

On Fifteen Love

A student demystifies the Baron's game of cards.

In their most recent game, Sir R----- was challenged to pick cards from the ace to nine of hearts so as to play a trick of three cards that summed to fifteen, counting the ace as a one, taking turns so picking with the Baron. If Sir R----- were to manage to do so before the Baron and before the cards were exhausted, he should have had a prize of one coin, forfeiting one if he weren't.

The simplest way to figure whether Sir R----- should have taken on the Baron is to arrange the cards in a magic square.



As can be plainly seen, every row and column sum to fifteen, as do the diagonals, and so if Sir R----- could have picked all of the cards from any of these before the Baron, he should have won the game.

Unfortunately, this is exactly the same as a game of noughts and crosses, which every schoolchild knows cannot be won if the opposing player has their wits about them. Indeed, I explained as much to the Baron but I fear that he may not have entirely grasped its significance.

Given that a draw should have counted as a win for the Baron, I would most certainly have advised Sir R----- to decline the wager! ■

Baron M

Courtesy of www.thusspakeak.com

BARON M

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

Join the ACCU

visit www.accu.org for details

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

Organised Chaos

Pete Goodliffe explains why organisation is so important.

The secret of all victory lies in the organisation of the non-obvious.
~ Marcus Aurelius

Good software is like crime; it's more effective when it's organised.

I joke with my team that we need to maintain the 'illusion of progress'. Every day as we work together to craft awesome software, we all need to see that we are progressing. It's important, and motivational, to understand that we're achieving what we intend. Both so we know we'll deliver on schedule, and so we can feel proud of the new value we're adding.

We feel better about our work, and more motivated to continue when we see that we are making progress. In order to genuinely gauge that we *are* progressing, we must have a series of achievable, measurable tasks that can be 'ticked off' when done.

This is why the Pomodoro technique [1] works, a simple micro-level system which time boxes your effort. It helps you to focus on small steps with clear goals and clearly measured results. When each 25 minute pomodoro finishes you know whether you've achieved your goal or not. On a grander scale this is also why processes like Scrum and XP work so well.

These stepwise approaches fulfil a basic human need – to achieve, to move forward, to make useful progress, and to be *rewarded* for doing so. (Usually sufficient reward is just the joy of seeing that you have achieved your goal! Any more 'serious' rewards may actually be detrimental to the development process.)

We do these stepwise software shuffles at the grand ceremonial, macro, level. We need to organise our large scale development plans. And we do it daily at the micro level. We need to organise our minuscule day-to-day tasks, focusing our effort in the right direction.

How do you keep yourself organised each day?

Organisation is key

The most effective programmers are not just technical geniuses, but are also well organised. They can reason about complex technical problems, and can also reason about the complexities of organising how to construct the solution.

To be an effective programmer you must be able to organise yourself well.

I can trust you to build me a great software system if I can hand you a list of requirements and I know that you will turn that into a reliable set of software construction tasks, and then arrange to execute them all to a schedule that we agree.

Honestly, That's all there is to software development!

So why do we make it so hard? Because breaking things up into achievable, manageable chunks is part science and part art. It relies on taste and experience. And because being organised enough to arrange those into a realistic schedule is a real skill, and one that most boffins seriously lack. (Along with common sense.) Daily we are distracted by a plethora of other tasks that need to be done, with other projects and interests vying for our attention.

Good managers can help mitigate this, and will empower their team to organise well. As do good development processes. We can set ourselves up to succeed by fostering an environment and a set of working practices that naturally lead to good organisation and naturally help to track progress, like the ones already discussed.

We can set ourselves up to succeed by fostering an environment and a set of working practices that naturally lead to good organisation

This is true in the macro level when we look at the concert of tasks we need to coordinate to deliver a software project, and also at the micro level when I consider how I, as a programmer, will deliver the right value today.

This is something I am increasingly finding myself doing: working out better ways to stay organised so that I can fulfil the myriad demands placed upon me each day.

As increasing numbers of things crop up that I have to do, I find myself more and more often writing lists to keep track of those things and help to prioritise them. Yes, I've discovered *to-do lists*! Once you start seeing them, you realise that most everything you do is on a list in some form or other.

So let's look at this most simple, and effective, technique to remain organised: the humble to do list.

TODO: Write a list

t might seem obvious (even trivial) how to use such a list. But let's look at these things in detail to gain mastery over them.

What is it?

A list. Of things to do. Simple as that.

In this context, a list of things you intend to do. Usually a single list holds one category of things, for example 'things to do today'. If technology permits, it may be arranged in priority order.

Why?

To stay organised. To make sure you keep on top of what you need to do, and don't forget anything.

To help you become a better programmer.

How do you do it?

Whichever way causes you the least friction and hassle. There are plenty of 21st century digital mechanisms, but often the simplest and most effective is still to use good old fashioned pen and paper. An advantage of paper is that it's always there and doesn't need to be switched on. It never runs out of batteries (although occasionally a pen will run out of ink! (And I admit that I lose my pen more often than I lose my computer.) Also, there's a lot to be said for the *tangible* experience of writing things down; it helps you remember and consider each item a little more.

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

Standards Report

Jonathan Wakely reports from the latest C and C++ meetings.

Hello, standards fans! For a change this column includes a first-hand report from the C committee, WG14, as I was able to attend their recent meeting in person. I'll start with a summary of the last C++ meeting, as that happened first.

The C++ committee met in Jacksonville, Florida, at the start of March. As usual it ran for six days (and nights!) with over 100 people attending. One of the key topics on the agenda was what features are going to be in C++17. As we get closer to 2017 we need to stop adding new features and polish what's already there, so the evolution working groups decided which proposals should be forwarded for review by the Core and Library working groups at the next meeting. That means we should be getting close to feature complete, and can send out a draft for international ballot later this year.

Some of the big decisions about C++17 content related to the Technical Specifications (TS) that have been published in the last few years, as experimental extensions to C++. At the start of the meeting Herb Sutter asked us to consider five publications for possible inclusion in C++17 (as well as the other proposals worked on during the week), and at the end of the week we decided on each one:

- The Mathematical Special Functions standard, which includes most of the extra maths functions from library TR1 that didn't get included in C++11. When we discussed this in Lenexa last year there were objections to requiring all implementations to support

JONATHAN WAKELY

Jonathan's interest in C++ and free software began at university and led to working in the tools team at Red Hat, via the market research and financial sectors. He works on GCC's C++ Standard Library and participates in the C++ standards committee. He can be reached at accu@kayari.org

Organised chaos (continued)

However, there are plenty of apps and tools that you may find more convenient. Digital versions offer a number of advantages – you can maintain a history of all your lists, make it easier to keep multiple lists (perhaps separated by topic, date, priority, frequency, etc), sync your lists between multiple devices, and more.

Popular Apps include: Apple's Reminders (and Notes), Google Keep, and a plethora of third party apps like Evernote and Wunderlist. This is a popular category so there's plenty to choose from. Find something that works for you. I'd recommend avoiding too many gimmicks, and selecting an app that is not distracting but offers goods functionality.

Personally, I use more than one medium. I use simple pen and paper when I want to make notes quickly and consume them as quickly (often when in meetings), and Apple Notes for longer lived tasks that are useful to keep synced between my phone and computer. As my daily to do list is digital, then things I failed to achieve (heaven forfend!) can easily roll over to the next day.

When?

I refer to my lists frequently throughout the day to determine what to do next. I keep lists of short-term goals as well as list for my longer-running missions.

At the start of each day I compile a todo list for the day: starting with the tasks I want to get done and the code I want to write. Then I read my email, and adjust it with any new work items that occur. Then I check my calendar and see if I've forgotten any appointments. If so, I'll add those in too. (Oh, the curse of meetings!)

Since I compile this in Apple Notes (other list-making apps are available) I can now prioritise the things that have to be done first. I take a realistic look at what this should perhaps be considered 'stretch goals' and set my expectations accordingly. I move items up and down in the list to reflect these priorities.

By now I have built myself a thoughtful plan for my day's activities.

Throughout the day as I work through these items, I tick them off on the list. The tick-off ceremony creates a feedback loop, rewarding me for making progress. Dopamine release!

I'll know if I've achieved my goals at the end of the day by looking at my list. If most of my list is ticked, I can go home feeling rewarded that I did what I set out to that day. (Of course, if other things side-tracked me and I achieve few of my planned tasks, I save to soul-search whether this was my fault for being easily distracted, or just a Day From Hell.)

You can use to do lists for many other organisational reasons.

They are particularly useful when you're working on something and want to capture a thought before it distracts you. The poster child for this is during TDD: to jot down ideas you can't focus on now, but you don't want to forget (e.g. new tests to write). I usually write a todo list of tests as comments at the bottom of my test file, rather than in a separate note taking application.

How to do it most effectively

Keep disciplined making (and reading) lists. Remember to do it.

Craft achievable, granular tasks. Set realistic expectations for what you can achieve.

Is it a panacea?

No, a to do list is clearly not the perfect planning tool. But as such a simple, low ceremony way of organising plans and thoughts it really is something worth considering picking up into your daily routine.

There are plenty of other (more complex) organisational techniques that might help you keep on top of tasks, for example: Personal Kanban [2] and GTD [3].

Conclusion

Effective developers are organised developers. You won't achieve anything worthwhile without a well-conceived plan. Making a plan doesn't have to be super-complicated. Your should employ every simple planning technique available to ensure you work as well as you can.

Now if only I could use the same techniques to get organised enough to wrote my column on time... Chance would be a fine thing. ■

References

[1] The Pomodoro Technique http://pomodorotechnique.com
[2] Personal Kanban http://personalkanban.com
[3] Getting Things Done http://gettingthingsdone.com

this, as the functions are not widely used outside of science and engineering, but in the end there was clear consensus for including it in C++17.

- The Parallelism TS, which adds new overloads of most of the algorithms that originally from the STL, allowing them to be executed in parallel (for example using thread pools, or SIMD vector instructions). The committee decided to include these in C++17 as well.
- The first Library Fundamentals TS, which adds lots of utility types including **`string_view`**, **`optional`**, **`any`**, and polymorphic allocators. Most of the TS content will be in C++17, but some parts were left out because they have not yet been implemented anywhere or because they would have introduced incompatible changes to existing pieces of the standard library.
- The Filesystem TS, based on **Boost.Filesystem**, was also voted into C++17.
- The Concepts TS, which extends the language with one of the biggest changes to happen in C++ recently (see Andrew Sutton's article in *Overload* 131 for more detail). There were strong opinions on both sides of the debate, but in the end the consensus was that the specification in the TS is not yet ready for the standard. There are some concerns about having multiple different ways to say the same thing, and that there is only one implementation which hadn't shipped yet (GCC 6 includes support for concepts and should have been released by the time you read this).

The other big feature which wasn't approved for inclusion in C++17 is unified call syntax, a proposal to allow **`f(x, a, b)`** to be used as an alternative syntax for **`x.f(a, b)`**. The main objections to the proposal were that it introduced a new kind of overloading as a second-class citizen, and it was felt that it should be better integrated into the existing overloading rules.

The new features which have been approved for C++17 include changes to lambdas so they can be used inside constexpr functions, and a new form of lambda capture to allow capturing **`*this`** by value (in C++ today a lambda defined inside a member function can only capture individual member variables by value, or capture them all by reference via capturing **`this`**). There are also three new attributes, including one to inform the compiler you really did mean to fall-through a switch case without a **`break`**, which means you can ask the compiler to warn you about any fall-through that doesn't use the attribute. That helps address what I consider one of the biggest flaws in C and C++, that the default behaviour in **`switch`** statements is to fall-through rather than to **`break`**.

Forwarded from the evolution group to be considered at the next meeting (so likely to be in C++17), are proposals to specify the order of evaluation in expressions (so that function arguments will be evaluated left-to-right), allowing operator-dot to be overloaded, and allowing class template arguments to be deduced from a constructor invocation, so that **`pair(1, '2')`** would be equivalent to **`pair<int, char>(1, '2')`**.

Although the current focus is getting C++17 feature-complete, work also continues on ranges, modules, coroutines, contracts and networking, which are all aiming for their own TS.

So that was C++ in Jacksonville, but in April I also attended the London meeting of the C committee, which was held at the BSI headquarters in Chiswick. Much of the time was spent processing recent defect reports, including several about underspecification of the new multithreading facilities in C11. If you look closely at the C11 standard you'll notice that it supports recursive mutexes, but doesn't say anything about how they work! It's also not clearly stated how **`mtx_init()`** behaves if you try to initialize a mutex more than once, or fail to initialize it at all before trying to lock it. Resolving defects like that isn't too hard, as nearly everyone agrees what the semantics should be, so it's just a case of writing the wording to specify it clearly and unambiguously.

Another topic of the meeting was the C memory model, based on attempts by Peter Sewell and his group at Cambridge who have been trying to come up with formal models of how the C abstract machine is meant to work according to the standard, and comparing that with how practising programmers think the language works. Among the problems they've found when comparing theory and practice are the behaviour of uninitialized data (what the C standard calls 'unspecified values'), whether there are any guarantees about the values of padding bytes between members of a structure, and the topic of 'pointer provenance', which is a similar issue to the object lifetime rules that the C++ committee were considering in Kona last year (see my last column).

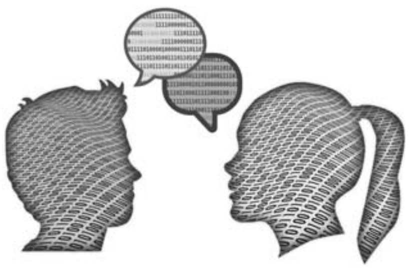
The committee also looked at proposals for new features, which might be considered for inclusion in a future C2x revision of the standard. Those included enhancements to what is valid in an integer constant expression (which is much more restrictive in C than in C++, even before the addition of **`constexpr`** in C++11), allowing the underlying type of an enumeration to be fixed (a very useful feature, which C++ allows since C++11), and a proposal to add a new **`__VA_OPT__`** feature to the preprocessor, to be used alongside **`__VA_ARGS__`** to solve problems that arise from using empty argument lists with variadic macros. As I've mentioned previously in this column, the preprocessor specification is largely the same in C and C++, so that last proposal is being considered jointly by the C and C++ committees, and so the hope is that they'll both agree on the same change, not two incompatible ones!

It's interesting to see C thinking about adding things that C++ already has in some form, and the last thing I'll mention also falls into that category. There is a proposal to add closures to C, based on the Apple 'Blocks' extension supported by C and Objective C compilers for Mac OS X and iOS. There is considerable overlap with C++ lambda expressions, but without references, templates, type deduction and other C++ features the proposal for C is necessarily different, especially the syntax. The feature would need non-trivial compiler and runtime support, which is not something typically expected of C compilers, especially those for small embedded systems, so it was no surprise that not everyone on the committee was in favour. The proposal wasn't rejected though, and so is likely to proceed as a TS.

That's all for this time, I hope I've covered most of the interesting topics. The next C++ meetings will be in Finland, in June, and near Seattle in November. The next C meetings will be in Pittsburgh in October, then Markham, Ontario, next April.

Code Critique Competition 99

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



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

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

Last issue's code

> I am writing a simple program to index written text but it doesn't quite work. I want to print out every word in the document with a list of each line it appears on. I'm only getting the first occurrence listed but can't work out why.

Please explain why they have this problem... and suggest some other possible improvements to the program. The program is in Listing 1.

Listing 1

```
#include <iostream>
#include <map>
#include <sstream>
#include <vector>
int main()
{
  using namespace std;
  map<string, vector<int>> index;
  // read and index standard input
  string line;
  int lineno{};
  while (getline(cin, line))
  {
    ++lineno;
    istringstream iss(line);
    string word;

    while (iss >> word)
    {
      auto start =
        word.find_first_not_of(":;.,'\"?!-");
      auto end =
        word.find_last_not_of(":;.,'\"?!-");
      if (start != end)
        word.replace(end + 1, end, "");
        word.replace(0, start, "");
      if (word.empty()) continue;
      auto iter = index.find(word);
      if (iter == index.end())
      {
        index[word].push_back(lineno);
      }
      else
      {
        auto lines = iter->second;
        if (lines.back() == lineno)
          ; // ignore dups
        else
          lines.push_back(lineno);
      }
    }
  }
```

Listing 1 (cont'd)

```
  // print the index
  for (auto entry : index)
  {
    cout << entry.first << ": ";
    string delim;
    for (auto line : entry.second)
    {
      cout << delim << line;
      delim = ", ";
    }
    cout << '\n';
  }
}
```

Critiques

Paul Floyd <m >

At first I thought that this was a fairly simple problem and that there would be very little to say.

Let's start with the reason why only the first occurrences are being detected. This is due to the code that handles the detection of the second or subsequent occurrence:

```
auto iter = index.find(word);
if (iter == index.end())
{
  index[word].push_back(lineno);
}
else
{
  auto lines = iter->second;
  if (lines.back() == lineno)
    ; // ignore dups
  else
    lines.push_back(lineno);
}
```

Specifically, **`auto lines`** infers a vector, so it makes a copy of the existing vector containing one entry. It then either does nothing or appends a line number to the end of this copy. The copy goes out of scope at the end of the closing brace, leaving the original unchanged.

This can be fairly easily fixed by making **`lines`** a reference. Simply declaring it as **`auto& lines`** will do the trick, or alternatively using an explicit declaration:

```
using MyMap = std::map<std::string,
              std::vector<int>>;
```

then

```
MyMap::mapped_type& lines = iter->second;
```

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

I don't like the use of the variable **`index`**. **`index`** is so commonly used to mean loop index that I'd prefer to see something that doesn't overload the meaning, like **`word_index`**.

The next code smell was the punctuation detection. This looks like it should be in a function. The choice of punctuation seems fairly arbitrary. Personally I would use **`ispunct()`**, and only use something different if necessary. There are a couple more things that I would avoid:

- **`start`** and **`end`** variable names, too confusing when there are functions and members with the same name in the standard library
- **`replace`** with an empty string; **`erase`** does the same thing and is more idiomatic (at least if you think of **`std::string`** as being a container rather than a string)

I would write this as:

```
const char* myPunct = ":;.,'\"?!-";
word.erase(0, word.find_first_not_of(myPunct));
word.erase(word.find_last_not_of(myPunct)+1);
```

This avoids the need for local variables for the start and end of the string.

James Holland <James.Holland@babcockinternational.com>

The student has made a fair attempt at the design of this program. However, there are several problems which prevent the software working as required.

- Braces should surround the two statements following the first **`if`** statement. Both **`replace()`** functions need to be dependant on the **`if`** statement, not just the first as is the case with the student's code. To avoid this type of error it is best to get into the habit of always enclosing conditional parts of an **`if`** statement in braces even if they consist of only one statement.
- The first statement of the **`else`** block of the third **`if`** statement defines a variable named **`lines`**. This should be a reference type so that the vector within the map can be modified. As it stands, additional line numbers are added to a local copy of the **`lines vector`** and not the **`lines vector`** within **`index`**.
- The program does not select words that are separated only by the delimiting characters (**`":;.,"\?!-"`**) and not a white space. Presumably, the program should interpret a line containing **`"name::space"`**, for example, as two words. Unfortunately, this problem requires some fairly major restructuring of the code to solve and will be discussed later.

In addition, there are some issues that are more to do with style that the correct working of the program. I consider them worth discussing, nonetheless.

- The delimiting characters have been defined twice. It would be better to declare a **`const string`** that is initialised to the required value and used instead of the repeated literal strings. Any required change to the delimiting characters need then only be done in one place.
- The student makes use of **`string`**'s **`replace()`** member function. This is confusing because it gives the impression that some characters of the **`string`** are being replaced by other characters. What is really happening is that characters are being erased. It would be better to use one of the overloaded **`erase()`** functions to clarify the code.
- The variables **`entry`** and **`line`** that are used to print the index need not be copies; they need only be **`const`** references. This is unlikely to make a significant speed increase when writing to the console, but it does make it clear that no modifications are being made to **`index`**.

As mentioned above, the student's program does not isolate words that are only separated by the delimiting characters. It may well be the case that there are many such words in a line of text. This suggests that a loop is required that isolates the words and only exits when there are no more in the line. Writing the code for a loop (in this case a **`while`** loop) can be tricky but after a little thought and some trial and error, I came up with the following program.

```
#include <iostream>
#include <map>
#include <vector>
int main()
{
  using namespace std;
  const string delims(" \t:;.,'\"?!-");
  map<string, vector<int>> index;
  string line;
  int line_number{};
  while (getline(cin, line))
  {
    ++line_number;
    auto start_of_word =
      line.find_first_not_of(delims);
    while (start_of_word != string::npos)
    {
      auto end_of_word =
        line.find_first_of(delims,
          start_of_word);
      if (end_of_word == string::npos)
      {
        end_of_word = line.length();
      }
      const string word(line.substr(
        start_of_word,
        end_of_word - start_of_word));
      auto position_of_word = index.find(word);

      if (position_of_word == index.end())
      {
        index[word].push_back(line_number);
      }
      else
      {
        auto & lines = position_of_word->second;
        if (lines.back() != line_number)
        {
          lines.push_back(line_number);
        }
      }
      start_of_word = line.find_first_not_of(
        delims, end_of_word);
    }
  }
  for (const auto & entry : index)
  {
    cout << entry.first << ": ";
    string delim;
    for (const auto & line_number :
      entry.second)
    {
      cout << delim << line_number;
      delim = ", ";
    }
    cout << '\n';
  }
}
```

As well as correctly selecting words from the line of text, this program has the advantage that there is no need to 'top and tail' words using **`replace()`** (or **`erase()`**). Also, experiments show that the revised program is about 20% faster that the student's (ignoring inputting the data and outputting the results).

Commentary

While it is very nice to have two keen and regular supporters of the code critique, can I encourage you to have a go even if you've never entered the competition before? You can see your name in print and it is good practice for real code reviews!

There were, as Paul noted, a fair number of problems in a relatively simple-looking piece of code…

I think between them the entrants covered most of the points pretty well. The original presenting problem was due to naive use of **`auto`**. The design principle to bear in mind is that C++ uses value semantics in very many places (see for example Andrzej Krzemieński's blog post at https://akrzemi1.wordpress.com/2012/02/03/value-semantics/). Hence the default behaviour of plain **`auto`** is to make a new *value* even when the original item is a reference. For reasons that are unclear to me, this behaviour seems unintuitive, at least initially, to many programmers who assume the compiler will give them the same type they would have written without the presence of **`auto`**.

When I use **`auto`**, I find myself writing **`auto const &, auto *`**, etc., a significant proportion of the time to either enforce or highlight (or both) the semantics of the generated variable.

The final bug was that the first call to replace uses the wrong value for the second argument (the number of characters to erase) – the code is currently written as:

```
word.replace(end + 1, end, "");
```

but the actual number of characters that need to be removed is from position **`end+1`** to the end of the string. However, passing **`end`** to the replace function will not cause any undefined behaviour, it just may not remove *enough* characters in some pathological cases. James' solution side-steps this problem completely as using **`erase`** means the number of trailing characters does not need to be supplied.

The winner of CC 98

Both critiques were good but I think James covered a bit more ground, and also uncovered the design flaw that the original program does not handle embedded delimiters, so he wins the prize for this issue's critique.

Code critique 99

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

> I wanted to learn a bit about C++ threading so I tried writing a thread pool example. But it sometimes crashes – I've managed to get it down to a small example. Sometimes I get what I expected as output, for example:
>
> ```
> Worker done
> Worker done
> Ending thread #2
> Ending thread #0
> Worker done
> Ending thread #1
> Worker done
> Ending thread #3
> Worker done
> All done
> ```
>
> But other times I get a failure, for example:
>
> ```
> Worker done
> Ending thread #0
> Worker done
> Awaiting thread #1
> Worker done
> W
> <crash>
> ```
>
> I'm not sure what to do next – can you help?

The program is in Listing 2.

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 2

```
#include <algorithm>
using namespace std;
#include <array>
#include <chrono>
using namespace chrono;
#include <cstdlib>
#include <iostream>
#include <thread>

static const int POOL_SIZE = 4;

// Allow up to 4 active threads
array<thread, POOL_SIZE> pool;

// Example 'worker' -- would in practice
// perform some, potentially slow, calculation
void worker()
{
  this_thread::sleep_for(
    milliseconds(rand() % 1000));

  cout << "Worker done\n";
}
// Launch the thread functoid 't' in a new
// thread, if there's room for one
template <typename T>
bool launch(T t)
{
  auto it = find_if(pool.begin(), pool.end(),
    [](thread const &thr)
    { return thr.get_id() == thread::id(); }
  );
  if (it == pool.end())
  {
    // everyone is busy
    return false;
  }
  *it = thread([=]()
  {
    t();
    thread self;
    swap(*it, self);
    self.detach();
    cout << "Ending thread #"
      << (it - pool.begin()) << "\n";
  });
  return true;
}

int main()
{
  while (launch(worker))
  {}
  // And finally run one in this thread as an
  // example of what we do when the pool is full
  worker();

  for (auto & it : pool)
  {
    thread thread;
    swap(thread, it);
    if (thread.joinable())
    {
      cout << "Awaiting thread #"
        << (&it - &*pool.begin()) << "\n";
      thread.join();
    }
  }
  cout << "All done\n";
}
```

Bookcase

The latest roundup of book reviews.

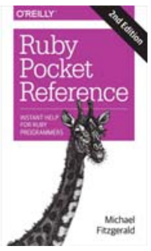
If you want to review a book, your first port of call should be the members section of the ACCU website, which contains a list of all of the books currently available. If there is something that you want to review, but can't find on there, just ask. It is possible that we can get hold of it.

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

Thanks to Pearson and Computer Bookshop for their continued support in providing us with books.

Astrid Byro (astrid.byro@gmail.com)

Ruby Pocket Reference 2nd Edition



By Michael Fitzgerald. (ISBN 978-1-491-92601-7, £9.99, published by O'Reilly, 216 pages, Index, Glossary)

Reviewed by Ian Bruntlett

I bought this book for two reasons. One, to act as an aide-mémoire when coming back to Ruby after programming in other languages. Two, to act as a concise overview of the language whilst I also make my way through the more detailed work, *The Ruby Programming Language*.

So, what does this book cover? I'll give a brief rundown here. It covers using Ruby and supporting software (Ri, Rake, RubyGems), the language in general and an introduction to some key Ruby library references. I found the glossary to be useful and the list of Ruby resources (books and websites) to be very useful as well.

A few things I noticed. In parts it has references to the ruby-doc website for more information – quite useful. It would have been helpful if it had a) mentioned the basics of debugging Ruby programs b) a separate index for method names and c) if the Ruby Operator's table listed the associativity of the operators as well.

On the whole, this book did what I expected it to do. This book is essentially a springboard into Ruby programming but needs to be complemented by other works.

Cloud Computing Design Patterns

By Robert Cope, Thomas Erl and Amin Naserpour. Hardback. 540pp. Published by Prentice Hall. ISBN 0-13-385856-1

Reviewed by Alan Lenton

First a word of caution. Although it's not clear from the book's ambiguous title, the patterns in this book relate to solving common hardware problems in setting up data centres for providing cloud services, not software design patterns for programs running in the cloud.

Having said that, the book does cover most of the basics, although some of the material could be argued to fall into the category of the blindingly obvious! As material on patterns should, it documents best practice in solving common problems in cloud data centres. Unfortunately, the technology in this industry is moving very rapidly, while publishing continues to move at the same snail's pace it did when I was running a bookshop over 30 years ago. This means there are two major omissions – containerisation à la Docker and its ilk, and Software Defined Networking (SDN).

A few years ago I might have recommended this book for those moving toward what was then a very embryonic version of dev-ops, but now the absence of containerisation and SDN material makes it unsuitable for this role.

ACCU London

Mrs Trellis reviews the March 16th meeting, without a clue.

Dear speaking clock,

I recently attended ACCU London's meetup "Party Like it 2015 with C# 6" given by Jon Skeet, "The Chuck Norris" of programming [1]. He neither counted to infinity twice, nor recited the digits of π backwards, which is for the best since we therefore managed to go to a local pub afterwards. Instead he demonstrated various C#6 features, using Noda time [2] which is no surprise, but decided to use Comics Sans throughout, which was something of a surprise. Live coding instead of slides was a refreshing change.

The room was packed, though there were a few chairs left and Jon Skeet asked why there were so few women. I shall therefore invite all my female friends next time.

He covered various features (probably all the new features) include the null-conditional operator "?.", auto-property initialisers which tidy up read only-fields, nameof expressions and various other things. As he talked through string interpolation C# seems to have decided to try to be more like perl, though he had a spot of trouble with his colon at this point. I sympathise. He may have a new version of his *In Depth* book out covering C#6 and 7 in the future.

I assume the time will be 16:50 precisely at the next beep?

Yours sincerely,

Mrs Trellis,
North Wales.

References

[1] http://www.bbc.co.uk/news/uk-england-34596634

[2] http://blog.nodatime.org/