the magazine of the accu

www.accu.org

{cvu}

Volume 31 • Issue 4 • September 2019 • £4.50

Features

Regulars

Key partners include:

Agisoft · ASPOSE File Format APIs · ATLASSIAN · axure · BrowserStack · DevExpress · embarcadero · Hex-Rays state-of-the-art code analysis · intel Software · JetBrains Platinum Reseller · Microsoft Silver Partner · qbs PUBLISHING · Sketch · SMARTBEAR · SPARX SYSTEMS · Telerik Develop experiences · think-cell · Visual Paradigm

Plus many more on **www.qbssoftware.com/developer**

For your latest software needs, contact our team on:

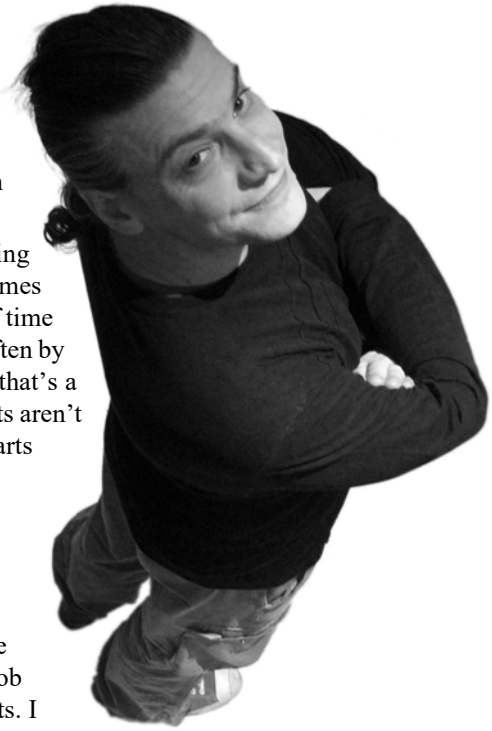
020 8733 7100 sales@qbs.co.uk

qbs SOFTWARE

{cvu}
Volume 31 Issue 4
September 2019
ISSN 1354-3164
www.accu.org

accu

This is only a test



It sometimes seems to me that the idea of automated software testing hasn't really taken hold. I often hear of development teams talking up the fact that they do Test Driven Development, but when pressed, admit the tests aren't really *driving* the effort, and are often pretty sparse.

I also hear of teams making a big thing of being really 'test focussed', but when pressed, it becomes apparent they mean they spend huge amounts of time and effort in manually testing their software – often by just running it to see what happens. Of course, that's a valid, even important aspect of testing – unit tests aren't great at testing whether an application even starts successfully – but it should be one of many aspects, rather than the whole game.

A common refrain is that developers don't have time (or aren't paid) for writing tests, they need to spend their time on writing productive code. Related to this are those developers who insist that it's someone else's job (usually, a junior team member) to write the tests. I suspect most readers of *CVu* would agree that this misses the vital point about test code, that *it isn't just test code*. It's part of the whole conversation about the code, documenting the author's assumptions and approaches, both good and bad, as well as providing a facility to check for regressions in the future. Unit tests are a form of communication, not merely an extra thing to do, or box to tick.

Higher quality tests provide higher quality communications within a team, and to future generations of the team working with those tests, and the code they support. There is so much more to tests than testing!

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 31.5: 1st October 2019
C Vu 31.6: 1st December 2019

Overload 153: 1st November 2019
Overload 154: 1st January 2020

WRITE FOR C VU

Both *CVu* 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

Making a Linux Desktop

Alan Griffiths illustrates how to get started with Mir.

I'm working on a project (Mir) that, among other things, aims to make it easy to develop graphical 'desktop environments' for Linux.[1] There are a lot of features that are common between all designs for desktop environments and, in addition, a lot that are common between the majority of designs. For example, it is common for applications to draw 'windows' and for these to be combined onto the screen.

By providing the common features, and for the rest offering both sensible defaults and an easy way to customise them, Mir is designed to support a range of possible designs.

Last year, I wrote an article for *CVu* [2] about a simple graphical shell. Since then, there have been improvements in the 'out of the box' facilities offered by the Mir libraries and I'm revisiting the subject.

This is the first of in a new series of articles that illustrate the development of an Example Mir Desktop Environment. [3]

Wayland

Most current desktop environments for Linux are based on X11. This is the last in a line of protocols for communicating between the applications, and parts of the 'desktop environment'. However, the environment in which it works has changed somewhat since the 1980s. That means many fundamental design decisions needed to be re-evaluated.

The result of this re-evaluation is a new protocol: 'Wayland'. This has been implemented for a number of application 'toolkits' and desktop environments. Instead of being based on X11, Mir supports the newer Wayland protocol.

Preparation

The code in this article needs Mir 1.2 (or later). On Ubuntu 18.04 (and later) this is available from the mir-team/release PPA. It is also useful to install the weston package as the example makes use of weston-terminal as a Wayland based terminal application and the Qt toolkit's Wayland support: qtwayland5. And finally, the g++ compiler and cmake.

```
$ sudo apt-add-repository ppa:mir-team/release
$ sudo apt install libmiral-dev mir-graphics-
  drivers-desktop
$ sudo apt install weston qtwayland5
$ sudo apt install g++ cmake
```

At the time of writing, Mir 1.2 isn't available on Fedora. But I hope that will be resolved by the time you read this:

```
$ sudo dnf install mir-devel
$ sudo dnf install weston qt5-qtwayland
$ sudo dnf install gcc-c++ cmake
```

Mir is also available from the Arch AUR and can be built from source for many versions of Linux.

Egmde

This series of articles follow the development of a very simple desktop environment. Because this is an example, don't expect it to be polished to a production level. But it is usable and could be the basis for further work.

Building the example

The full code for this example is available on github:

```
$ git clone https://github.com/AlanGriffiths/
  egmde.git
$ git checkout Article-1
```

Naturally, the code is likely to evolve, so you will find other branches, but the Article-1 branch goes with this article. Assuming that you've MirAL installed as described above, you can now build egmde as follows:

```
$ mkdir egmde/build
$ cd egmde/build
$ cmake ..
$ make
```

Running the example

After this you can start egmde:

```
$ ./egmde
```

You should see a black Mir-on-X window with a cursor. You can launch a terminal window with Ctrl-Alt-T (or, if that conflicts with your desktop, Ctrl-Alt-Shift-T).

You should see the same black screen with a weston-terminal window. From this you can run commands and, in particular, start graphical applications. Perhaps qtcreator to examine the code?

You should find that you can do all the normal things – use the keyboard and mouse to switch between and resize windows. If you have a touchscreen, then that will work too.

Running a full desktop type session is also possible, but first I should mention that it is possible to close egmde using the keyboard combination Ctrl-Alt-BkSp. Now switch to a virtual terminal (Ctrl-Alt-F4 for example) log in and run egmde-desktop.

Installing the example

If you install egmde, it will be added to the greeter menu (typically a 'cog wheel' on the sign-on screen):

```
$ sudo make install
```

Depending upon the greeter used on your system, it may be necessary to reboot before egmde appears as a login shell.

Installing the example

To register applications with desktop environments we install a `.desktop` file in `/usr/share/applications` and to register Wayland based desktops with the greeter we install a `.desktop` file in `/usr/share/wayland-sessions/`. The content needed for both cases is similar, so for this example we use the same `.desktop` file and install it to both places.

The example code

A lot of the functionality (default placement of windows, menus etc.) comes with the MirAL library. This means that there's very little code needed to get this basic shell running:

```
$ wc -l *.cpp
75 egmde.cpp
```

The main program for this article is shown in Listing 1 (overleaf).

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

Listing 1

```
using namespace miral;
int main(int argc, char const* argv[])
{
  MirRunner runner{argc, argv};
  ExternalClientLauncher
external_client_launcher;
  auto const keyboard_shortcuts
      = [&](MirEvent const* event)
  {
    if (mir_event_get_type(event)
        != mir_event_type_input)
      return false;
    MirInputEvent const* input_event
        = mir_event_get_input_event(event);
    if (mir_input_event_get_type(input_event)
        != mir_input_event_type_key)
      return false;
    MirKeyboardEvent const* kev
        = mir_input_event_get_keyboard_event
        (input_event);
    if (mir_keyboard_event_action(kev)
        != mir_keyboard_action_down)
      return false;
    MirInputEventModifiers mods
        = mir_keyboard_event_modifiers(kev);
    if (!(mods & mir_input_event_modifier_alt) ||
        !(mods & mir_input_event_modifier_ctrl))
      return false;
    switch (mir_keyboard_event_scan_code(kev))
    {
      case KEY_BACKSPACE:
        runner.stop();
        return true;
      case KEY_T:
        external_client_launcher.launch
          ({"weston-terminal"});
        return true;
      default:
        return false;
    }
  };
  return runner.run_with(
  {
    set_window_management_policy
    <MinimalWindowManager>(),
    external_client_launcher,
    AppendEventFilter{keyboard_shortcuts},
    Keymap{},
  });
}
```

This breaks down into three blocks, the first is:

```
MirRunner runner{argc, argv};
ExternalClientLauncher external_client_launcher;
```

These are two objects from the MirAL library. The 'runner' takes care of running Mir; it takes the command line arguments to handle configuration options. The `external_client_launcher` is used later to launch the terminal window.

The second block is a lambda – `keyboard_shortcuts` – that does some basic input handling. For Ctrl-Alt-T it starts a terminal using `external_client_launcher`; for Ctrl-Alt-BkSp it tells runner to stop the server.

The final block is a call to `runner.run_with()`, which takes a list of customizations, applies them to the server and runs it. This is the main way of customizing your server (we'll be looking at other options next time). The customizations used here include the `keyboard_shortcuts` and `external_client_launcher` mentioned above. There are also a couple more MirAL library objects: `Keymap` and `MinimalWindowManager` which respectively manage the keyboard layout and provide some default window management.

The MirAL API

The 'Mir Abstraction Layer' is an API designed to make it easy to use Mir. It provides a way to use Mir that avoids the complexities, and other issues, associated with the earlier mirserver API. (The development of the MirAL API is described in *Overload* 136, December 2016 [4].)

The API can be split into the `MirRunner` and a number of classes that are passed to the `MirRunner` to provide customizations. The customizations seen here are for launching external clients, filtering input, finding the current keyboard layout and setting the window manager.

From this point it is possible to extend the 'desktop' in various ways – for example, the `MinimalWindowManager` can be replaced with one that implements 'tiling' (there's an example in the Mir code that does this).

Conclusion

This article shows how the Mir library makes it easy to get started with a limited, but working 'desktop environment'.

The next article will continue from this point and add features such as a 'wallpaper' background and a 'launcher' for selecting applications. ■

References

[1] The Mir homepage: https://mir-server.io/

[2] Alan Griffiths (2018) 'Writing a Wayland Server Using Mir' in *CVu* 30.2 May 2018

[3] The egmde git repo: https://github.com/AlanGriffiths/egmde

[4] Alan Griffiths (2016) 'The MirAL Story' in *Overload* 136 December 2016

Attitude, Accomplishment, Artistry

Pete Goodliffe looks at our attitude to the code we write.

From caring comes courage.
~ Lao Tzu

It doesn't take Sherlock Holmes to work out that good programmers write good code. Bad programmers... don't. They produce elephantine monstrosities that the rest of us have to clean up. You want to write the good stuff, right? You want to be a good programmer.

Good code doesn't pop out of thin air. It isn't something that happens by luck when the planets align. To get good code you have to work at it. Hard. And you'll only get good code if you actually *care* about good code.

> To write good code, you have to *care* about it. To become a better programmer, you must invest time and effort.

Good programming is not born from mere technical competence. I've seen highly intellectual programmers who can produce intense and impressive algorithms, who know their language standard by heart, but who write the most awful code. It's painful to read, painful to use, and painful to modify. I've seen more humble programmers who stick to very simple code, but who write elegant and expressive programs that are a joy to work with.

Based on my years of experience in the software factory, I've concluded that the real difference between mediocre programmers and great programmers is this: *attitude*. Good programming lies in taking a professional approach, and *wanting* to write the best software you can, within the real-world constraints and pressures of the software factory.

The code to hell is paved with good intentions. To be an excellent programmer you have to rise above good intentions and actually *care* about the code – foster positive perspectives and develop healthy attitudes. Great code is carefully crafted by master artisans, not thoughtlessly hacked out by sloppy programmers or erected mysteriously by self-professed coding gurus.

You want to write good code. You want to be a good programmer. So, you care about the code. This means you act accordingly; for example:

- In any coding situation, you refuse to hack something that only *seems* to work. You strive to craft elegant code that is clearly correct (and has good tests to show that it is correct).
- You write code that *reveals intent* (that other programmers can easily pick up and understand), that is *maintainable* (that you, or other programmers, will be able to easily modify in the future), and that is *correct* (you take all steps possible to determine that you *have* solved the problem, not just made it look like the program works).
- You work well alongside other programmers. No programmer is an island. Few programmers work alone; most work in a team of programmers, either in a company environment or on an open source project. You consider other programmers, and construct code that others can read. You want the team to write the best software possible, rather than to make yourself look clever.
- Any time you touch a piece of code, you strive to leave it better than you found it (better structured, better tested, and more understandable...).
- You care about code and about programming, so you are constantly learning new languages, idioms, and techniques. But you only apply them when appropriate.

Fortunately, you joined ACCU because you *do* care about code. You're reading this magazine because it interests you. It's your passion. You like coding *well*. Keep working at it; turn code concern into practical action.

As you do this, never forget to have fun programming. Enjoy cutting code to solve tricky problems. Produce software that makes you proud. ■

> There is nothing wrong with an emotional response to code. Being proud of your great work, or disgusted at bad code, is healthy.

Questions

- Do you *care* about code? How does this manifest in the work you produce?
- Do you want to improve as a programmer? What areas do you think you need to work on the most?
- If you don't care about code, why are you reading this magazine?!
- How accurate is the statement *Good programmers write good code. Bad programmers... don't*? Is it possible for good programmers to write bad code? How?

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

A Case Against the Use of Code 'Smells'

Simon Sebright asks if we're using the term correctly.

I recently saw a few tweets about deleting code. I am all for that, the first tweet was resounding the good feeling you have when you delete a lot and in this case replace it by a lot less, but that does not have to be the case, sometimes you just get rid of stuff that is not needed.

Anyway, a tweeted reply said something like, 'I love the smell of deleted code'. Hmm, yes, I imagined summer meadows, mountain air, hoppy beer, and stuff like that.

But we are constantly confronted with the concept of 'code smells', a plural noun, not a sentence. So that implies that smells are a bad thing. I'd like to recapture the neutrality of the word smell. It means that our sense of smell can detect something, perhaps interpret it. It does not imply bad stuff. In English I would choose the word 'stink' for that. The farmer has been muck-spreading. It stinks. Sure it smells too, but so do my roses. They don't stink.

So, let's use 'Bad code smells' and 'Good code smells' to indicate what we mean. Or code 'stenches' for bad things (but what about good – fragrances???). Clean code smells like a spring morning, or lavender, or your shampoo. Bad code smells like dung, mould, the dungeon in a castle still used to house prisoners.

But, let's conclude by conceding that all metaphors have their limits (otherwise they would not be metaphors). Interestingly, humans cannot imagine smells in the way that (most) can imagine a shape or a sound. I can certainly imagine good and bad code, so maybe we need to find another metaphor. ■

SIMON SEBRIGHT

Simon has been in Software and Solution development for over 20 years, with a focus on code quality and good practice. He can be reached at simonsebright@hotmail.com

The Standards Report

Guy Davidson provides his latest report.

In this report, I'm going to cover the new language proposals that were voted into the standard at the recent meeting in Cologne. Twenty-five motions came before the committee from the Core Working Group (CWG) for voting. Rather than iterate through them all, I shall present some highlights. Recall that any paper can be reviewed by visiting wg21.link and appending the proposal number; for example, https://wg21.link/P1000.

The biggest news from CWG was P1823, 'Remove contracts from C++20'. It is very hard to remove things from the International Standard. Once code is written against the standard, any change which renders that code non-conforming will carry considerable political impact. Engineers are faced with the choice of either rewriting their code or freezing their C++ implementation at a point prior to this change. Neither of these is particularly palatable, and they both weaken confidence in the language and the committee. Therefore, if a feature turns out to be incorrectly specified, no matter how popular it is, it needs to be withdrawn before it creates greater damage outside of the committee.

In the case of contracts there were a number of problems, as outlined in the motivation of the paper:

> Contracts were added to the working draft in Rapperswil 2018 with the usual assumption that we have agreement on the general goal and programming API of contracts. It turned out that this is not the case. In fact, we currently face the following situation:
>
> - We had major design changes proposed and accepted by EWG this meeting (Cologne, July 2019)
> - We have significant disagreement whether these changes make things better or worse. In fact, even the initial authors of Contracts for C++20 have 4o agreement.
> - We still discuss what the design changes mean.
> - We still design details on the fly.
> - Some major concerns were raised regarding the proposed changes.
> - We have no implementation experience (such as applying the currently proposed feature in the standard library).

Is this the end of contracts? Not at all. A new study group, SG21 Contracts, chaired by John Spicer of Edison Design Group, was formed to finish contracts. I wish them the best of luck.

Having mentioned how tricky it is to withdraw features from the standard, I must mention P1152, 'Deprecating volatile'. The title is a little deceptive: this isn't a wholesale removal of the keyword from the language. In particular, from the wording, we see that:

> Postfix ++ and -- expressions ([expr.post.incr]) and prefix ++ and -- expressions ([expr.pre.incr]) of volatile-qualified arithmetic and pointer types are deprecated.
>
> Certain assignments where the left operand is a volatile-qualified non-class type are deprecated; see [expr.ass].
>
> A function type ([dcl.fct]) with a parameter with volatile-qualified type or with a volatile-qualified return type is deprecated.
>
> A structured binding ([dcl.struct.bind]) of a volatile-qualified type is deprecated.

Many clauses describing the use of volatile remain unchanged. Take a look at the paper for more information on how we got to this point, particularly in the earlier revisions.

An interestingly innocent-looking motion that was passed was P1161, 'Deprecate uses of the comma operator in subscripting expressions'. Particularly, in this example:

```
void f(int *a, int b, int c) {
  a[b,c];   // deprecated
  a[(b,c)]; // OK
}
```

This suits me very well: I am co-authoring a linear algebra proposal and this allows us to use the comma operator for multidimensional subscript operators, essential for classes such as matrix.

The mission to 'constexpr all the things' continues apace with four excellent proposals. P1331, 'Permitting trivial default initialization in constexpr contexts', proposes permitting default initialization for trivially default constructible types in constexpr contexts while continuing to disallow the invocation of undefined behavior. So, as long as uninitialized values are not read from, such states should be permitted in constexpr in both heap and stack allocated scenarios.

P1668R1, 'Enabling constexpr Intrinsics By Permitting Unevaluated inline-assembly in constexpr Functions', does what it says on the tin: constexpr functions may now contain assembly.

My favourite of these, P1143R2, 'Adding the `constinit` keyword', saves us from the Static Initialisation Order Fiasco. Static storage duration variables with dynamic initializers cause hard-to-find bugs caused by the indeterminate order of dynamic initialization. By qualifying them with the `constinit` keyword they are initialised at compile time, on the condition that their constructors are constexpr. I am STILL caught out by the SIOF even after thirty years of C++. I still see it in code reviews, and it simply isn't going away. This mitigation is most welcome.

Finally, P0784, 'More constexpr containers', enables more containers to be constexpr. It achieves this by introducing constexpr destructors, constexpr new-expressions and enabling the use of the default allocator in constant expressions. This is going to be important for the upcoming reflection and metaprogramming API: all of the compile time introspection will require variable sized containers, currently impossible without these features.

The [[nodiscard]] attribute also gained some weight. P1301, '[[nodiscard("should have a reason")]]', delivers the potential for improved compiler diagnostics. From the paper:

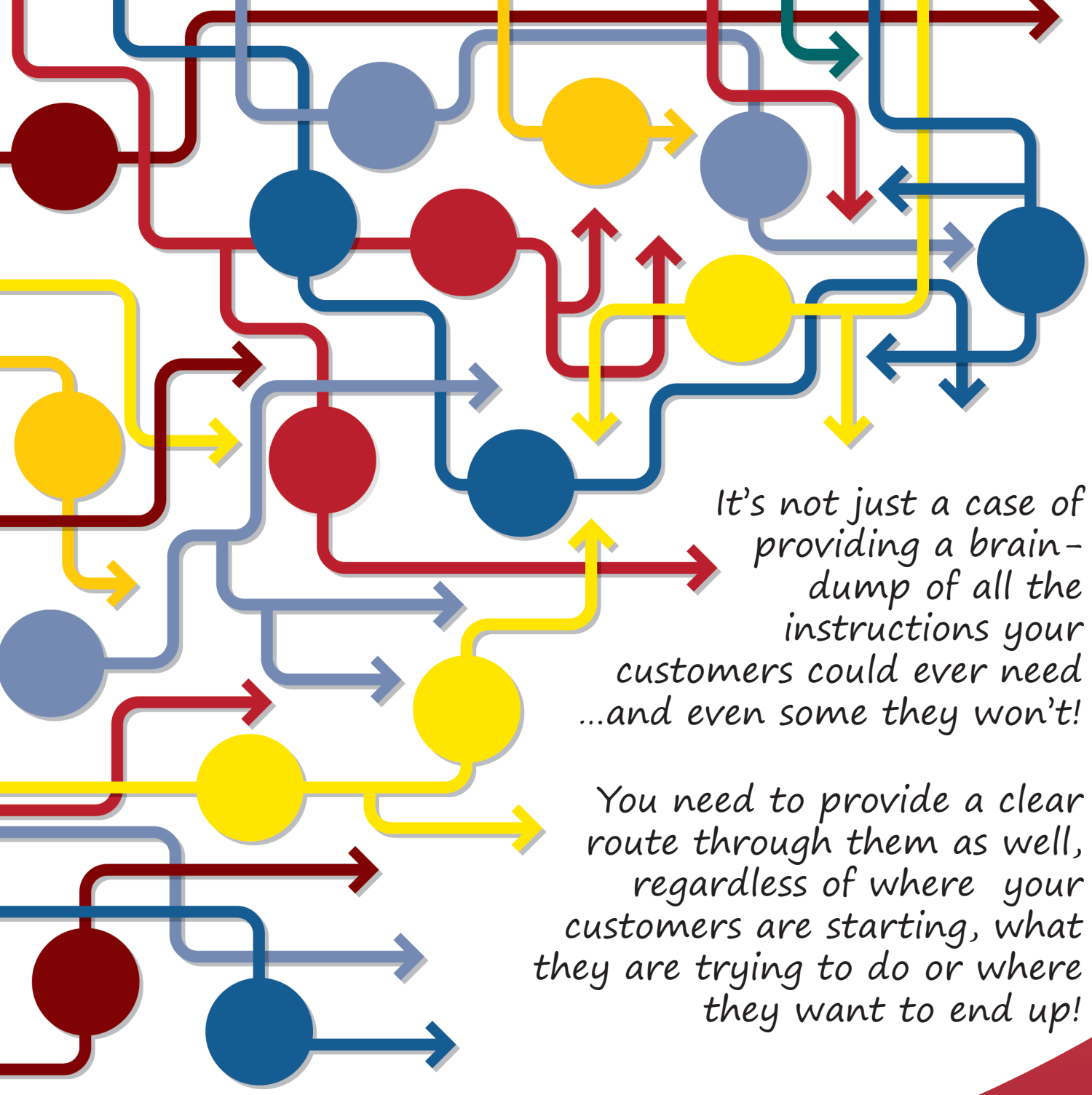
> The [[nodiscard]] attribute has helped prevent a serious class of software bugs, but sometimes it is hard to communicate exactly why a function is marked as [[nodiscard]].
>
> This paper adds an addendum to allow a person to add a string attribute token to let someone provide a small reasoning or reminder for why a function has been marked [[nodiscard("Please do not leak this memory!")]].

Finally, there was an interesting procedural point for P1714R1 (NTTP are incomplete without float, double, and long double!). This paper was brought to the committee after the feature freeze, and although it is a good paper with strong motivation, we had to decide if we could allow it in at this late stage. By now, proposals should either be reviewed and completed work, or bug fixes. The question was asked of the implementers present in the room: do you think this is a bug-fix? They unanimously answered "No". We voted to discard the motion to add this proposal: we voted not to vote. This averted setting a precedent that is unhelpful to our process. I look forward to this proposal entering the next Working Draft at the appropriate time, which I expect will be at closing plenary of the Summer 2020 meeting in Bulgaria.

In my next report I will discuss the library changes that were made in Cologne.

GUY DAVIDSON

Guy Davidson is the Principal Coding Manager at Creative Assembly. He has been writing games for about 75% of his life now. He is climbing through his piano grades and teaching Tai Chi while parenting in Hove and engaging in local political activism.



It's not just a case of providing a brain-dump of all the instructions your customers could ever need ...and even some they won't!

You need to provide a clear route through them as well, regardless of where your customers are starting, what they are trying to do or where they want to end up!

Get in touch for an informal discussion on how we can help you improve your user documentation:

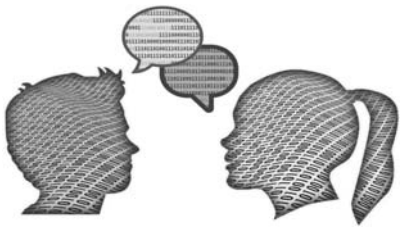
Clearly stated...

T 0115 8492271

E enquiries@clearly-stated.co.uk

W www.clearly-stated.co.uk

Code Critique Competition 119



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

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

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

Last issue's code

I'm writing code to process sets of address lists, and was hoping to avoid having to make copies of the data structures by using references. However, I can't get it working – I've produced a stripped-down example that reads sample data from a file but it doesn't work at all.

Sample data:

```
Roger Peckham London UK
John Beaconsfield Bucks UK
Michael Chicago Illinois USA
```

Expected output:

addresses, sorted by country and then county

Actual output:

three blank lines!

Can you solve the problem – and then give some further advice?

Listing 1 contains the code.

Listing 1

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <map>
#include <string>
#include <vector>
struct address /* simplified */
{
  std::string city;
  std::string county;
  std::string country;
};
std::istream& operator>>(std::istream& is,
  address& t)
{
  is >> t.city >> t.county >> t.country;
  return is;
}
std::ostream& operator<<(std::ostream& os,
  address& t)
{
  os << t.city << ' '<< t.county << ' '
     << t.country;
  return os;
}
// sort addresses by country, county, and
// then city
bool operator<(address const &a,
               address const & b)
{
  if (a.country < b.country) return true;
  if (a.country == b.country &&
      a.county < b.county) return true;
  if (a.country == b.country &&
      a.county == b.county &&
      a.city < b.city) return true;
  return false;
}

// This is just for testing, real data
// is supplied differently
auto read(std::istream &is)
{
  std::map<std::string, address> ret;
  while (is)
  {
    std::string name;
    address addr;
    if (is >> name >> addr)
    {
      ret[name] = addr;
    }
  }
  return ret;
}

int main()
{
  std::map<std::string, address> map;
  map = read(std::cin);

  // Don't copy the addresses
  std::vector<std::tuple<address&>> addrs;
  for (auto entry : map)
  {
    addrs.push_back(entry.second);
  }

  // sort and print the addresses
  std::sort(addrs.begin(), addrs.end());
  for (auto& v : addrs)
  {
    std::cout << std::get<0>(v) << '\n';
  }
  std::cout << '\n';
}
```

Listing 1 (cont'd)

Critiques

Nicolas Golaszewski <golaszewski.nicolas@gmail.com>

The Original Code (OC) wanted to spare copies by using references, which unfortunately cannot be used inside `std::vector`. Using a `std::tuple` with only one reference as the vector's element was a nice

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

trick. However, when it comes to sorting, this also brings in the use of swapping. Given **a** and **b** (two values of type **T**) and **Ra** and **Rb** (respectively the two references), the normal first step of swapping:

```
T & Rc = Ra;
```

does not create a copy (of **a**) but just a synonym of **Ra**.

The second swapping step

```
Ra = Rb;
```

ends by getting rid of the value of **a** in favor of the value of **b**.

So, why not use pointers to save copies? Old time raw pointers are now discouraged. Smart **std::shared_ptr<T>** would maybe be too expensive when a copy of the **T** value is cheap.

Fortunately there exists **std::reference_wrapper<T>** which just... encapsulates a raw pointer to **T**.

So, just rewording the OC **addrs** variable

```
std::vector<std::tuple<std::reference_wrapper
<address>>> addrs;
```

is enough to make things work again.

Other ideas on the OC came from the operator

```
bool operator<(address const &a,
  address const & b){...}
```

which defines the comparison order by country first, county and city last.

This is exactly what the **operator<** for **std::tuple** does.

So, the idea is to recycle the tuple trick to hold the 3 members of the address (I know the **address** class is simplified but I have seen the comparison **operator<** manually redefined many times where a tuple would get the job done).

So, the idea is to have an empty **address** class just to define what kind of data will be used

```
struct address  {
  // only good idea if sorting order stays
  // the same
  enum item { country = 0, county, city };
  using type = std::tuple<std::string,
               std::string,
               std::string>;
  using ref_wrap_type =
    std::reference_wrapper<type>;
  using ref_wrap_const_type =
    std::reference_wrapper<const type>;
};
```

And, for example, the input **operator>>** becomes

```
std::istream& operator>>(
  std::istream& is,address::type& t)
{
  is >> std::get<address::city>(t)
     >> std::get<address::county>(t)
     >> std::get<address::country>(t);
  return is;
}
```

The most interesting thing is in the comparison **operator<** – we do not need **operator<** for **address** as we will use the **operator<** of **std::tuple**.

The **auto read(std::istream &is)** testing function stays the same.

The **main()** code becomes

```
//...
std::map<std::string, address::type> map =
  read(ifs);

// Don't copy the addresses
std::vector<address::ref_wrap_const_type>
  addrs;

for (const auto & entry : map) {
  addrs.push_back(entry.second);
}
//...
```

Note that now the **map** value type is a tuple of 3 **std::string** and that the **addrs** vector holds reference to this **const** qualified type. (Put **const** whenever possible...)

I thought this would be the end of the story but, naturally, the **std::sort** algorithm will look for a comparison **operator<** of… **std::reference_wrapper<address::type>**… which naturally (and fortunately) does not exist.

Not being discouraged, the following adjustment with a lambda as comparator

```
std::sort(addrs.begin(), addrs.end(),
       [](address::ref_wrap_const_type rwc_a,
          address::ref_wrap_const_type rwc_b)
  {return rwc_a.get() < rwc_b.get();}
);
```

was simple enough to escape this pitfall.

Below is the complete code, tested on coliru just in case…

```
g++ -std=c++14 -O2 -Wall -pedantic main.cpp && ./a.out
```

```
#include <algorithm>
#include <iostream>
#include <map>
#include <string>
#include <vector>
#include <tuple>
#include <fstream>
#include <functional>
#include <sstream>
struct address
{
  // only good idea if sorting order stays
  // the same
  enum item { country = 0, county, city };
  using type = std::tuple<
    std::string,std::string,std::string>;
  using ref_wrap_type          =
    std::reference_wrapper<type>;
  using ref_wrap_const_type    =
    std::reference_wrapper<const type>;
};
std::istream& operator>>(
  std::istream& is,address::type& t)
{
  is >> std::get<address::city>(t)
     >> std::get<address::county>(t)
     >> std::get<address::country>(t);
  return is;
}

std::ostream& operator<<(
  std::ostream& os,const address::type& t)
{
  os << std::get<address::city>(t) << ' '
     << std::get<address::county>(t) << ' '
     << std::get<address::country>(t);
  return os;
}

/*We do not need operator< for address as we
  will use the operator< of std::tuple*/

// This is just for testing, real data
// is supplied differently
auto read(std::istream &is)
{
  std::map<std::string, address::type> ret;
```

```
  while (is) {
    std::string name;
    address::type addr;
    if (is >> name >> addr) {
      ret[name] = addr;
    }
  }
  return ret;
}
int main()
{
  std::stringstream  ifs(
    "Roger Peckham London UK\n"
    "John Beaconsfield Bucks UK\n"
    "Bart Springfield Illinois USA\n"
    "Michael Chicago Illinois USA\n"
    "Fuzz Oldbury West_Midlands UK\n"
    "Lisa Springfield Illinois USA\n"
    "Thomas Champaign Illinois USA\n");
  std::map<std::string, address::type> map =
    read(ifs);

  // Don't copy the addresses
  std::vector<address::ref_wrap_const_type>
    addrs;
  for (const auto & entry : map)  {
    addrs.push_back(entry.second);
  }
  for (const auto& v : addrs)  {
    std::cout << v << '\n';
  }
  std::cout <<
    "\n--------------------------------\n\n";
  std::sort(addrs.begin(), addrs.end(),
        [](address::ref_wrap_const_type rwc_a,
           address::ref_wrap_const_type rwc_b)
        {return rwc_a.get() < rwc_b.get();});
  for (const auto& v : addrs) {
    std::cout << v << '\n';
  }
  std::cout << std::flush;
}
```

James Holland <James.Holland@babcockinternational.com>

The student states that the stripped-down example reads data from a file. In fact, it reads data from **std::cin**. This is a small matter but it means that the end-of-file character has to be entered from the keyboard by pressing the appropriate key. On my system this is Ctrl-D.

Having sorted the input, we can now investigate the processing of the data. The student states that three blank lines are produced. When I ran the program it issued "Peckham London UK" on three separate lines. I must confess I am not exactly sure why the program produces this output but it has something to do with declaring the vector **addrs** with a template parameter of type **std::tuple<address &>**. Removing the **&** results in the program working as expected. Now the tuple holds a copy of objects of type **address**. This is unfortunate as the student wanted to store references to **address** objects. The way the student has declared the tuple to store references seems perfectly natural but adding the **&** is incorrect. The correct way to store references is to declare the tuple without the **&** but to make a tuple from an **address** object by using **std::ref()** (declared in the functional header file). This requires an extra line in the **for**-loop as shown below.

```
  std::vector<std::tuple<address>> addrs;
  for (auto & entry : map)
  {
    auto tup =
      std::make_tuple(std::ref(entry.second));
    addrs.push_back(tup);
  }
```

I have also made the **entry** loop variable a reference. This may help speed things up as **entry** will no longer be a copy of **entry.second**. The whole tuple is copied, however, when it is pushed onto the back of the vector.

I notice that **operator<()** has been defined explicitly. There is a slightly neater way of defining **operator<()** by making use of **std::tie()** as shown below.

```
bool operator<(address const & a,
  address const & b)
{
  return std::tie(a.country, a.county, a.city)
    < std::tie(b.country, b.county, b.city);
}
```

In this version of **operator<()**, the two addresses are first decomposed into the component parts and then reassembled into tuple objects containing three objects: the country, the county and finally the city. When comparing two tuple objects, initially the first parameters of the tuples are compared. If these parameters are equal, the next two parameters are compared, and so on. This is exactly the behaviour that is required for comparing countries, counties, and cities.

As the code uses **std::tuple**, the header file, **#include <tuple>**, should be added explicitly. In the student's program, I suspect the tuple header file is being found in **#include <map>**. The second parameter of **operator<<()** could be made **const** as could the identifier **entry** of the first range-based **for**-loop of **main()**. There is no need to abbreviate variable names quite so much, I suggest. Perhaps the variable **addrs** could be renamed **addresses_of_candidates** or something similar that is appropriate to the application.

The desire to write fast and efficient code is understandable but it is important to, firstly, ensure the code works correctly. Only then is it worth considering modifications that are designed to make the code run faster. Compilers are getting very clever at optimising code for speed and so one should always measure the performance of any code modification to see if it has really made an improvement. I know of cases where an enthusiastic engineer has amended code with the desire of making it faster but has actually made it run slower. It has been said that you should not help the compiler. If you keep things simple the compiler will know what you are trying to do and will probably make a good job at optimising the code. This does not mean that you should not provide the compiler with as much information as possible, however. As an example, making function parameters **const ref** where appropriate is almost always a good idea.

As the code provided is a stripped-down version of the original application, it is hard to know why particular features have been employed. I am thinking particularly of the use of **std::tuple**. I can think of no use for a tuple that contains just one value but this may be a result of providing example code that exhibits the problem. Perhaps the code could be designed not use **std::tuple** thus simplifying things further.

Marcel Marré < marcel.marre@gerbil-enterprises.de> and Jan Eisenhauer <mail@jan-ubben.de>

The main issue of this competition's code is the use of references. The original author writes in a comment that he doesn't want to copy the addresses as he builds the vector to be sorted. However, **for (auto entry : map)** does exactly that: it makes a copy. Even worse, it makes a temporary copy that is thrown away later, pushing us into the realm of undefined behaviour, which can explain the three empty lines (in my g++ I got three identical lines rather than empty lines; this kind of different behaviour across implementations can indicate undefined behaviour).

However, even after changing the loop to **for (auto & entry : map)**, all is not well. One entry is duplicated, another gone. The problem is again an incorrect use of references. The original author stores the addresses to be sorted in a **std::vector<std::tuple<address&>>**. The page on **std::sort** on cppreference, https://en.cppreference.com/w/cpp/algorithm/sort, notes that the iterators must be ValueSwappable. This, however, does not hold for references. Rather than swapping references

in the vector, we change data in the map that the vector entries refer to. In other words, with the `std::tuple<address&>`, construction has different semantics from assignment.

With `std::reference_wrapper`, construction and assignment have consistent semantics, rebinding the reference in both cases. Hence, changing the vector to `std::vector<std::reference_wrapper <address>>` corrects the code's behaviour.

Following the rule of being as `const` as possible, we can also declare the output operator as `std::ostream& operator<<(std::ostream & os, address const & t)` with the body remaining unchanged. This also goes for the output loop: `for (auto const & v : addrs)`.

While this fixes the behaviour of the code, the `operator<` can also be written a lot more elegantly in modern C++:

```
return std::tie(a.country, a.county, a.city)
  < std::tie(b.country, b.county, b.city);
```

is the whole required body of the operator, which is immensely more readable, more easily expanded for further fields and thus also a lot less error-prone.

Even though the `read` function is only for testing, it can also be abbreviated, because formatted input on a bad stream has no effect:

```
auto read(std::istream &is)
{
  std::map<std::string, address> ret;
  std::string name;
  address addr;
  while (is >> name >> addr)
  {
    ret[name] = addr;
  }
  return ret;
}
```

This also reuses the capacity of the strings in each iteration of the `while`-loop rather than constructing a new local string each time.

Finally, the code uses nothing from the `<iterator>` header, so it is cleaner to remove the include.

Hans Vredeveld <accu@closingbrace.nl>

When I build (with g++ 7.4.0 on Linux) and execute the program with the same data as the OP, I don't get three blank lines, but three times the line 'Peckham London UK'. It looks like we run into some undefined behaviour.

Before we delve into the main problem, let's get some small issues out of the way.

The include for `tuple` is missing.

There is a spurious include of `iterator`.

Like the function read, the `operator>>` to read in an address is fine for the testing done here, but it is too simple to handle real data. For example, consider the Dutch city Den Haag in the province Zuid Holland.

The `operator<<` to output an address, does not modify the address and should take it by `const&`.

To improve readability, it would be nice to have a class or typedef `name_t` and write `std::map<name_t, address>`, instead of writing `std::map<std::string, address>`.

In `main`, using `map` both for the type and for the variable is confusing. I suggest renaming the variable to `addr_map`.

The first two statements in `main` can be combined into a single statement that initializes the map on construction with the result of `read`.

The `for` loop to print the addresses at the end of `main` should take an address as `const&: for (auto const& v : addrs)`.

With the small things out of the way, we can start to look into the main issues with the code. In the following, it is instructive to also look at the content of the address map. Therefore, let's add the following code snippet to the end of main:

```
for (auto const& entry : addr_map)
{
  std::cout << entry.first << " -> "
    << entry.second << '\n';
}
std::cout << '\n';
```

This will show the input data sorted on name.

The OP wanted to avoid copying addresses. Unfortunately (s)he forgot that when you don't explicitly specify the loop variable in a range-based `for`-loop as reference, it will take the elements of the container by copy. The result is that the vector `addrs` is filled with references to a local variable that is out of scope by the time we sort the vector. In my test situation, it still contained the data of the last address added.

Making the loop-variable entry a reference, the elements in the vector now reference the elements in the map. Building and executing the program, I now get the output

```
Beaconsfield Bucks UK
Chicago Illinois USA
Chicago Illinois USA

John -> Beaconsfield Bucks UK
Michael -> Chicago Illinois USA
Roger -> Chicago Illinois USA
```

In other words, by sorting the addresses, we made Roger move in with Michael. A serious breach of data integrity! (Replacing `std::sort` with `std::stable_sort`, I was able to let Roger move to Chicago and let John and Michael both move to Peckham. Other implementations of `std::sort` and `std::stable_sort` may have John, Michael and Roger move around between Beaconsfield, Chicago and Peckham in other ways.)

To use `std::tuple<address&>` as an element type for the vector `addrs` is a clever trick to store references in the vector. Too clever, in my opinion. Also note that we don't want to change anything in the map when sorting the vector, but that changing the `addrs`'s type to `std::vector<std::tuple<address const&>>` will not compile. To understand why, we have to delve a little bit into what is happening in `std::sort`. Whatever actual sorting algorithm is used, `std::sort` will have to move around the elements in the vector. It can only do this by first moving one or more elements into temporary variables, then move other elements into the free spaces in the vector and finally move the elements from the temporary variables into the (new) free spaces in the vector. For example, say that element 0 and 1 have to be swapped. This will result in something similar to:

```
auto temp = std::move(addrs[0]); // (1)
addrs[0] = std::move(addrs[1]);  // (2)
addrs[1] = std::move(temp);      // (3)
```

In (1), we move-construct a temporary variable of type `std::tuple<address&>`. This will initialize the embedded reference with `std::forward<address&>( std::get<0>(addrs[0]))`, which, because of reference collapsing, means that the reference is copied. Now both `temp` and `addrs[0]` reference the same address object. Next, in (2), we move-assign from `addrs[1]` to `addrs[0]`. This assigns `std::forward<address&>(get<0>(addrs[1]))` to `get<0>(addrs[0])`. Again, reference collapsing applies, and we have an assignment of one lvalue-reference to another lvalue-reference. This is nothing more than copying the content of the address object referenced by `addrs[1]` to the address object referenced by `addrs[0]`. Finally, in (3), like in (2), we copy the content of the address object referenced by `temp`, which was changed via `addrs[0]` in (2), to the address object referenced by `addrs[1]`. As a result, we changed the content of the objects referenced by `addrs[0]` and `addrs[1]` to both have the value that the object referenced by `addrs[1]` had before. Sorting the vector thus modifies the map.

To get the desired behaviour of sorting the vector of addresses, we have to use pointers instead of references in the vector. This will also remove the need to use `tuple`, and we can also make everything related to the map (and the map itself) `const`. In the `sort` statement, we have to add a predicate to sort on the values pointed to instead of the pointers themselves. `main` now has become

```
int main()
{
  std::map<name_t, address> const addr_map
    = read(std::cin);

  // Don't copy the addresses
  std::vector<address const*> addrs;
  for (auto const& entry : addr_map)
  {
    addrs.push_back(&entry.second);
  }

  // sort and print the addresses
  std::sort(addrs.begin(), addrs.end(),
    [](auto a, auto b) { return *a < *b; });
  for (auto const& v : addrs)
  {
    std::cout << *v << '\n';
  }
  std::cout << '\n';
}
```

Note that we can also use the algorithms `std::transform` to fill the vector from the map and `std::for_each` to print the addresses. This is left as an exercise for the reader.

Balog Pál <pasa@lib.hu>

This code looks really interesting. However, before engaging, let me insert the rant I have been pushing back for the last few entries and as I really can't take it any more. Iostreams. ☹ Can we say goodbye to them? It is only noise and distraction. If we still have related issues, what is the chance we have not pointed them out 36 times over in the 117 previous entries? Especially the input part that only fills our vector or map – all `std::` collections have very fine `{}` initialization, the input(s) should just be created that way. For outputs, I'd also prefer using some simple framework like catch, those who just read printed code can hopefully understand it without a peek.

Having said that, I ignore the `iostream` parts of this sample. Ok, except for the missing `const` in the `<<`, but that is it. With that eliminated we have just half the code. Not even half, just 4 effective lines. How many problems can we fit in this tiny space?

Certainly my improved version would not compile having the map `const` as it should be… and the series of changes needed just to compile would likely remove the problems too. So let's go somewhat more conservatively and leave the map mutable. While at it, mark the leading 2 lines for doing 'create-then-assign' instead of just initialize, and naming the map `map`… that is legal but not nice in my book.

A collection of references packed into a tuple is interesting to start with. Is that even valid? Let's put that question aside and for the time being assume it is. We iterate over the map incorrectly. Range-based `for` without `&` is suspect and the norm is to have `const&` there. Otherwise it creates a copy of each item, which works at gimped speed when we just use the value and leads to catastrophic failure if we happen to store the address. Which is our case? Looks like the second: we bind the address to the copy not the original as intended. As a side note, `emplace_back` is a more fitting call here. Even considering that in modern C++, we get the same code for both because of mandatory copy elimination.

Then we sort the vector. We didn't pass a compare function, so we'll need an `op<` here. Ah, we have it too, just lost between the `<< >>` noise. It looks bad whichever way you look at it, even with a quick glance. The current canonical form is simple: `return tie(lhs.k1, lhs.k2... ) < tie(rhs.k1, rhs.k2... )`. Yeah, no kidding. Maybe magical but will surely do what is needed, just arrange the same sequence of keys and no lhs/rhs typos once. If done by hand, then you take key one, return immediately if they are `<` or `>`, continue with the next key if neither. We miss the step right on the second line with a bad combination. With some luck, in a year's time we can just ask for a defaulted `<=>` and roll with it.

Messing up the `compare` function is no small deal either, as we face UB if it breaks the rules of irreflexivity, transitivity and transitivity of equivalence. I'm too lazy to follow up with the actual data; it might be benign by pure luck but don't count on it. Some STL implementations have nice instrumentation and might point out a problem for you, but again that is not a thing to rely on: just make sure your function is correct. If in doubt, ask an independent review. Some cases appear unintuitive, i.e. I have had to explain that "comparing doubles with tolerance is NOT legal in a sort function!" a few dozen times to the same set of people.

So, we have fixed the vector and the compare function, will we sort well now? Well, sort-of. ☺ Now it will sort something; the question is, how happy we are with that? As we use references in the vector, `sort` will swap around the values sitting in the map, slicing them off their original keys. So, we have the map itself ordered: John, Michael, Roger. And the addresses will be sorted too, as Bucks, London, Illinois, associated to the people based on that order. The last two guys may not be particularly enthusiastic about that.

Certainly, it would not be evident from the code that it outputs only the addresses, not bothering with the map yet.

With all problems addressed, it's time to get back to the initial issue of whether collecting and sorting references, that are after all immutable beings (yeah, a reference is not even an object), is a fair deal or not. I honestly cannot tell for sure. I followed the criteria in the standard and got swamped at 'equivalent', which is inconveniently not defined. We have a nice answer on SO: https://stackoverflow.com/questions/37142510/can-i-sort-vector-of-tuple-of-references where the user Barry is adamant that MoveAssignable this way is not conforming on this last point (formally everything seems valid, as the `type_traits` functions report `true`). He may be right. But if 'equivalent' means equality or substitutability, then with references it kinda works in its weird way – as we never observe the ref itself just the object behind it. As long as they all look at a distinct object.

Interesting that Berry pointed out `sort` as bad, but if it is, similar requirement break happens on the vector too.

I pass this mystery over to Roger who will hopefully go into detail on both how to interpret the situation and why it is not clear from the standard text – we might get a core issue here, or at least an editorial addition of a note to shorten the search for the next time

Commentary

This code is an example of someone knowing just enough to be dangerous. The most 'interesting' line is the declaration of `addrs`:

```
std::vector<std::tuple<address&>> addrs;
```

Why did they use a tuple with a single element? As Nicolas correctly stated, it's because you can't create a vector of references… but by wrapping the reference in a tuple you *can*. This critique partly answers the question of whether you *should*.

As several people commented, `std::reference_wrapper` is designed for this purpose: it wraps a reference in a copyable and assignable object. (Note that while James's solution correctly creates a `reference_wrapper` object by using `std::ref`, but as it is then immediately used to construct a copy the reference nature is not retained.)

References in C++ are quite tricky – they are, to a certain extent, 'invisible' as you can't really access a reference, it acts as what can best be described as an alias for an existing object.

So given `a` and `b` of type `int&` the expression a = b simply does an assignment of the referenced objects; and if the type of both changes to `std::tuple<int&>`, the assignment still assigns the referenced objects.

While references, and tuples of references, comply with the *syntax* of MoveAssignable (and `std::is_move_assignable_v<>` will be `true`), they do not comply with the expected *semantics*.

Generally, `const` references are relatively safe (as long as the lifetime of the referenced object outlasts the reference) but non-`const` references can be troublesome, especially as member data whether directly or, as in this case, indirectly via a template argument.

Finally, Pál's diatribe against iostream is partly answered, at least for output, by the merge of `std::format` into the working paper for C++20 at the recent WG21 meeting in Cologne. See http://wg21.link/P0645 for the formal wording. (We've also had several informal mentions of this in recent articles in *Overload*/*CVu*.)

The winner of CC 118

All the entrants successfully resolved the presenting problem. Several also noted that, for code that tried hard to avoid copies by using a 'trick' with a tuple, the range-`for` loop was *copying* each element. Marcel and Jan also noted that hoisting the strings `name` and `address` out of the loop in the `read` function would be a small performance win.

I like Hans' step-by-step explanation of what actually breaks when sorting the collection of references; I hope this explanation would make it clearer to the programmer not only what they had done wrong but why the results were what they were!

The implementation of `operator<` received some comments – you have to read it quite carefully to check whether it is correct. It does seem that the message about using `std::tie` to implement these methods is getting heard as almost all the solutions removed the hand-crafted code.

There were good answers from all entrant, but in my opinion, Nicolas wins, by a short head!

Code Critique 119

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

I'm relatively new to C++ and I'm trying to build up a simple hierarchy of different entities, where their name and id and a few more characteristics are held in an `Entity` base class; and specific subclasses, such as `Widget` in the code below, hold the more specialised data – in this case simulated by the member name `data`. I've been having problems with the program I'm writing crashing and so I've simplified it down to a short example which usually produces output like this:

```
Test
none
Another widget
<core dump>
```

Please can you suggest what might be causing the core dump? I've had to use `-fpermissive` with gcc (but not msvc) – but surely that's not causing the problem?

Listing 2 contains `Entity.h`, Listing 3 is `Widget.h` and Listing 4 is `example.cpp`. As always, there are a number of other things you might want to draw to the author's attention as well as the presenting problem.

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

Listing 2

```
#pragma once
class Entity
{
private:
  char* pName{ NULL };
  int idNumber{ 0 };

public:
  void Setup(char const* name)
  {
    pName = strdup(name);
  }

  virtual ~Entity()
  {
    delete pName;
  }

  // Delete the current object using the dtor,
  // and re-create as a blank object
  void Reset()
  {
    this->~Entity();
    new(this) Entity();
  }

  char *Name()
  {
    return pName ? pName : "none";
  }
};
```

Listing 3

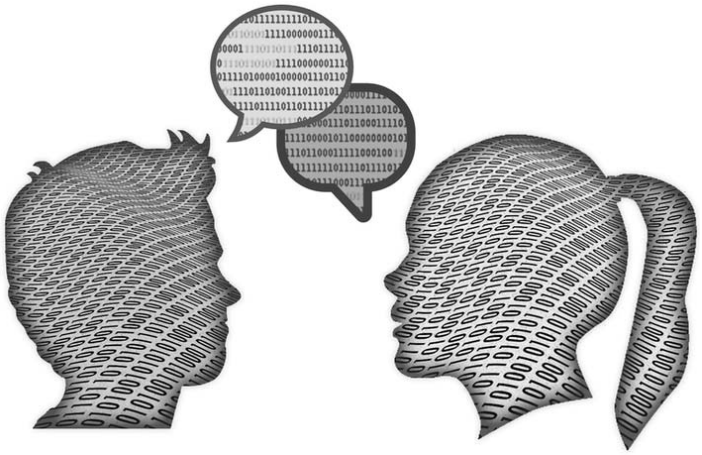
```
#pragma once
class Widget : public Entity
{
  int* data;
public:
  Widget() { data = new int[10]; }
  ~Widget() { delete data; }
};
```

Listing 4

```
#include <iostream>
#include <memory>
#include <string.h>

#include "Entity.h"
#include "Widget.h"

int main()
{
  Widget w;
  w.Setup("Test");
  std::cout << w.Name() << '\n';
  w.Reset();
  std::cout << w.Name() << '\n';
  w.Setup("Another widget");
  std::cout << w.Name() << '\n';
}
```



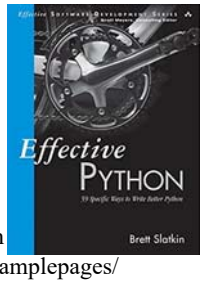
Reviews

The latest roundup of reviews.

We are always happy to receive reviews of anything that may be of use, or of interest, to software developers. The emphasis will tend to be on technical books, but we are also interested in less-technical works – for example the humanities or fiction – and in media other than traditional print books.

Do you know of something you think deserves – or does not deserve – a wider audience? Share it here. We're keen to highlight differing opinions on the same work, so if you disagree with a review here do send in your own! Contact Ian Bruntlett at reviews@accu.org.

Effective Python



By Brett Slatkin, published by Addison-Wesley, 2015, 216 pages, ISBN: 0-13-403428-7

Reviewed by Paul Floyd

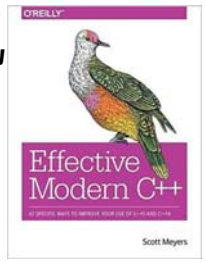
There is some sample content at: http://ptgmedia.pearsoncmg.com/images/9780134034287/samplepages/9780134034287.pdf

I'm only just starting with Python, so my rating (3/5) is perhaps a bit harsh as I don't think that this book is really intended for beginners. I've read a few other titles in the Addison-Wesley *Effective* series (*Code Quality*, **`Code Reading`** and *Effective XML*). Generally, I found them to be fairly easy reads with low barriers to entry. As with many books in this style, the book is organized into 59 'items' split over 8 chapters.

Effective Python jumps very quickly from a couple of introductory items (checking which Python version you are using, read the Python style guide) to some definitely *not* beginner items like array slicing. This meant that I found it easy to follow some of the easier items like docstrings but for items like decorators and metaclasses, it was all over my head and I could only imagine the kind of introspection that they provide.

I may revisit this review after going though the online python.org 'getting started' guide and perhaps reading a proper beginner's book.

Effective Modern C++



By Scott Meyers, published by O'Reilly, 2014, 301 pages, ISBN: 978-1-491-90399-5

Reviewed by Paul Floyd

Web page: http://shop.oreilly.com/product/0636920033707.do

Highly recommended (5/5).

Effective Modern C++ follows on in the tradition of the various other *Effective C++* volumes by the same author. The style is much the same, down to the use of colour in the code samples. This volume is also arranged into items, 42 of them in this case. Probably the biggest stylistic change is the switch from Addison-Wesley to O'Reilly, which means white-on-pink text and an animal (a rose-crowned fruit dove).

The book covers all of the major changes and additions that came with C++11 and C++14: **`auto`**/**`decltype`** type deduction, brace initialisation, **`nullptr`**, **`constexpr`**, class enums, **`default`**/**`override`**/**`delete`** for class methods, smart pointers, rvalue references, lambdas and concurrency. That's a lot of ground to cover and Meyers does a very good job of explaining it all in just over 300 pages. Not only did I find the explanations clear and easy to follow, but I also enjoyed the explanations of the benefits of using the new features. Meyers also makes it clear that the changes aren't always that much of a revolution; for instance, in the case of rvalue references he points out that the benefits are likely to limited due to the conservative choices that were made by the C++ committee on specifying when the compiler can automatically generate move constructors and move assignment operators.

I definitely felt upon finishing this book that I understood C+11/14 better.

The Design and Implementation of the FreeBSD Operating System, Second Edition

By Marshall McKusick, George Neville-Neil and Robert Watson, published by Addison-Wesley, 2015, 846 pages, ISBN: 978-0-321-96897-5

Reviewed by Paul Floyd

About 10 years ago, I read the first edition of this book, and I quite enjoyed it. I've read a few books on OSes (Solaris, Windows, Mac OS X and Linux) and they tend to dwell on the details, either in the case of the open-source OSes describing the fields of kernel data structures or in the case of the closed source ones, trying to find ways to tell you as much about the data structures as possible with kernel debug tools. My main recollection from the first edition was that there was little in the way of data structures, it was mainly English prose. And that I quite enjoyed.

Fast forward to the second edition. Firstly we jump from FreeBSD 5.2 to FreeBSD 11 (which is not yet a production release). A lot of changes have happened in those 10 years. Virtualization, Security, ZFS and DTrace have all been added, which probably explains why the page count has gone up by about 200 pages (though the paper is thinner so the second edition is actually slightly thinner).

Unfortunately for me, much of this new material was either fairly familiar to me (e.g. the stuff that came from OpenSolaris like ZFS and DTrace) or isn't really my cup of tea (like the security). That said, I quite enjoyed the book and if I do get the time to dabble a bit with FreeBSD then I expect that I'll revisit the book.

Boost C++ Application Development Cookbook

By Anthony Polukhin, published by Packt Publishing, 2013, 327 pages, ISBN: 978-1-84951-488-0

Reviewed by Paul Floyd

Covers Boost 1.53 (Note: 2nd edition published 2017.)

I've read a few books in the same mould as this, 'Boost recipe books'. They all share the same fundamental problem – why would someone buy the book when there is a similar amount of information available in the online Boost documentation. Boost includes well over 100 libraries, so consequently either thousands of pages are required, or else the coverage is either scant or incomplete.

Having said that, I quite liked the layout and the problem-oriented approach. In 327 pages, there just isn't enough information. Say I want to use Boost to generate some random numbers, there's a recipe for using **`Boost.Random`**. If I wanted to use a system generated uniformly distributed integer, then my luck is in. Otherwise I have to follow the link to the online Boost documentation.

In summary, it doesn't add much value compared to the online docs.

View from the Chair

Bob Schmidt
chair@accu.org

As I write this at the beginning of August, it's the dog days of summer and temperatures in Albuquerque have been hovering in the mid to upper 90s [1] to low 100s for over a month (not unusual for summer here). The summer monsoons [2], which help bring temperatures down (but unfortunately raise the humidity), have not been as consistent as they usually are this time of year. I empathize with those of you in Europe suffering through your own unaccustomed heat waves.

For those of you keeping track [5], this is my 20th View from the Chair (View). When the publication deadline approaches, I break out in a (metaphorical) cold sweat, wondering what I'm going to find to write about this time [6]. Fortunately, I get a lot of help.

Most of what I write about originates from our ACCU committee meetings. The committee meets every two months; we try to schedule the meeting so that it falls one or two weeks before the next *CVu* deadline. Scheduling a meeting can be a challenge. Meeting dates are complicated by the differing national holiday schedules of the committee members. The meeting time is complicated because the committee currently is composed of people from three different time zones – from UTC+2 (Central European Summer Time) to UTC-6 (U.S. Mountain Daylight Time).

(For the last three years, meetings have been held in what is the middle of the afternoon for most of our committee members, so that I don't have to be awake in the middle of the night. I realize that this is an inconvenience for those whose weekend afternoons get interrupted, and I want to take this opportunity to thank the rest of the committee for accommodating me.)

The process I follow when writing the *View* can best be described as chaotic [7]. I look back at my meeting notes, write out broad headings, and start filling in details. I bounce from one section of the *View* to another, trying [8] to organize my thoughts and write something coherent that you, the members, will find informative. I add, delete, move, and otherwise mangle words, sentences, and paragraphs in an attempt to create a coherent narrative.

Once I'm done [9] I send the copy out to the committee for comments, corrections, and criticism [10]. My fellow committee members are an excellent source of all three. Their feedback is invaluable, and I thank them for their assistance.

After one or more rounds of editing, the *View* goes to the production designer, and I get a couple of months to start thinking about the next one.

ACCU YouTube Channel

Videos from ACCU 2019 have been posted to our YouTube channel [11]. Jim Roper, Creative Director at Digital Medium, and his team have been creating and uploading the conference videos for several years now. They are responsible for the filming at the conference and all post-production and editing. The committee and I thank Jim and his crew for their continued great work on the videos.

ACCU Autumn 2019

This is my last chance to remind you that ACCU's two-day conference in Belfast, Northern Ireland is scheduled for the 11th and 12th of November [12]. Registration is open here [13].

Call for volunteers

As those of you who read this column regularly know [5], we have several chronically vacant committee positions. As an incentive to fill these positions, the committee is announcing a new policy. Anyone who volunteers for one of the positions the committee deems chronically vacant will qualify for a temporary membership deferral. If already a member, a new volunteer will have their next membership payment deferred for one month for each month of service, up to a year. If not currently a member, a new volunteer will be instated as a member for up to a year, as long as they continue in the role.

The positions the committee has determined are chronically vacant are: publicity, study groups, social media, and web editor. In addition, the position of treasurer will qualify for the incentive, in order to allow Rob Pauer to retire after many years of service to ACCU.

Please contact me if you are interested in one of these positions [14].

The author would like to apologize for the unusual level of snark and sarcasm this month. The heat has blasted away what few remaining impulse-control brain cells he may have had. Better hope that global climate change is indeed a Chinese hoax, or there may be more Views like this [15].

Notes and references

[1] Fahrenheit. This is the U.S. – as a general rule we don't do Celsius here.
[2] We get most of our moisture during the summer monsoon season [3], which describes a particular weather pattern that brings rain up from the Gulf of Mexico.
[3] Calling it the 'monsoon' season is giving it way too much credit. Albuquerque, New Mexico, being high desert, averages around 9.5 inches [4] of rain per year; recently that number has been closer to 6.5 inches. The monsoon season delivers about half of that yearly total.
[4] Yes, inches. We don't do (centi)metres, either.
[5] And if you're not, why not?
[6] I am green with envy at Fran Buontempo's ability to write her wonderful 'non-' editorials every two months.
[7] And that's being generous.
[8] And usually failing.
[9] For a given value of done.
[10] I'm expecting a lot of the latter this time!
[11] ACCU 2019 YouTube channel: https://www.youtube.com/channel/UCJhay24LTpO1s4bIZxuIqKw
[12] ACCU Autumn 2019 Conference: https://conference.accu.org/
[13] ACCU Autumn 2019 Conference Registration: http://www.cvent.com/events/accu-autumn-conf-2019-wg21/custom-18-8860107a2e864b698109640c823a1855.aspx?dvce=1
[14] Send inquiries to chair@accu.org.
[15] *And nobody wants that*.

We need your help!

ACCU is a volunteer organisation. Without volunteers, we cannot function. We need:

- Volunteers for vacant posts on the committee
- People to write articles (regularly or occasionally)
- People who can help out with particular short-term and long-term projects

If you would like to help but are not sure how you can – you may not have a lot of time, or may not be able to commit to anything long-term – please get in touch. You may have just the skills we need for a short-term project or to reduce the workload of another volunteer.


accu
professionalism in programming
BOOK YOUR PLACE NOW
H & W
H & W
NEW for 2019
accu Autumn conF 2019
2 days of engaging and up-to-the-minute content plus the opportunity to listen to influencial international speakers at the forefront of software.
CONFIRMED KEYNOTE SPEAKER
Herb Sutter
www.conference.accu.org @ACCUConf
11th-12th November 2019, Belfast Hilton Hotel

intel Software

CODE MAXIMIZED

from £510

#HighPerformance

Develop high performance parallel applications from enterprise to cloud, and HPC to AI using Intel® Parallel Studio XE. Deliver fast, scalable and reliable, parallel code.

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

To find out more about Intel products please contact us:
020 8733 7101 | sales@qbs.co.uk | www.qbssoftware.com/parallelstudio

qbs SOFTWARE