the magazine of the accu

www.accu.org

{cvu}

Volume 34 • Issue 6 • January 2023 • £4

Features

Regulars

“The magazines”

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

“The conferences”

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

“The community”

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

“The online forums”

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

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

ACCU | JOIN: IN

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

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

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

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

Design: Pete Goodliffe

{cvu}

Volume 34 Issue 6
January 2023

ISSN 1354-3164
www.accu.org

Editor
Steve Love
cvu@accu.org

Contributors
Guy Davidson, Francis Glassborow, Pete Goodliffe, Roger Orr, Christopher Preschern

Reviews
Ian Bruntlett
reviews@accu.org

ACCU Chair
Geoffrey Daniels
chair@accu.org

ACCU Secretary
Robin Williams
secretary@accu.org

ACCU Membership
Matthew Jones
accumembership@accu.org

ACCU Treasurer
PatrickMartin
treasurer@accu.org

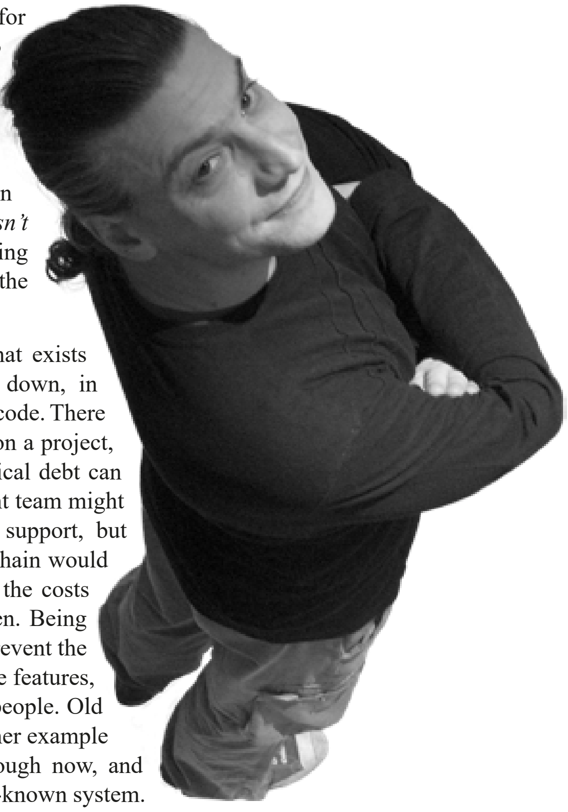
Advertising
[Vacancy]
ads@accu.org

Cover
Photo by Tim Peck – Shangri -La Hotel, Oman

Print and Distribution
Parchment (Oxford) Ltd

Original Design
Pete Goodlife

Technical Debt is a Burden



Technical debt is a term often reserved for problems in code: a new feature rushed into the codebase, perhaps with less than optimal testing, or a hasty design decision made before all the facts are known. When time is short, we accept the debt and promise ourselves that we'll revisit and improve it – pay it off – when we're in less of a rush. Sometimes that halcyon state *doesn't* happen, of course, and the debt lingers, accruing interest in most cases in the form of making the code harder to modify, maintain, or update.

Technical debt is something that something that exists in our development effort that's slowing us down, in whatever way. Not all technical debt lives in the code. There are many factors that can slow down progress on a project, but addressing them in the same way as technical debt can be harder to justify. For example, a development team might observe that their tools are dated or out of support, but updating to newer versions or a different tool chain would almost certainly be very disruptive. However, the costs of continuing with obsolete tools can be hidden. Being stuck with an old version of a compiler might prevent the team from taking advantage of modern language features, but may also be an impediment to hiring new people. Old or customized version control systems are another example where an existing system may work well enough now, and there's no business case to move to a more well-known system.

The problem with the 'no business case' argument is that the business (whoever they are) most often doesn't know or care what tools are used, or even what they're for, and will not suggest upgrading. The immediate benefits of upgrading are usually dwarfed by the cost and effort of doing so. What is less often considered is that obsolete tools are a form of technical debt, and are not only a continual burden on the team, but an increasing one, accruing interest along with all the other debt in the project.

Steve

STEVE LOVE
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

CVu 35.1	1st February 2023
CVu 35.2	1st April 2023
Overload 174:	1st March 2023
Overload 175:	1st May 2023

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 *CVu* or *Overload*, contact the advertising officer at ads@accu.org.

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

WRITE FOR US

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.

COPYRIGHTS AND TRADE MARKS

Some articles and other contributions use terms that are either registered trade marks or claimed as such. The use of such terms is not intended to support nor disparage any trade mark claim. On request we will withdraw all references to a specific trade mark and its owner.

The Human Touch

Pete Goodliffe looks at the personal side of programming.

> *Two things are infinite: the universe and human stupidity; and I'm not sure about the universe.*
> ~ Albert Einstein

Almost since the first programs were constructed we have realised that programming is not a solely technical challenge. It is also a social challenge. Software development is a pastime that involves writing code *with* other people, *for* other people to understand. It means working with other people's (variable quality) code, joining and leaving software teams, working under your boss' supervision, managing developers (which is rather like herding cats), and so on.

Programming is a People Pursuit.

Many of the most enduring programming tomes are devoted to the people problem, for example: *The Mythical Man Month* [1], and *Peopleware* [2].

Just as the people working with a codebase will inevitably shape the code they produce, the people who work with you will inevitably shape *you*.

Purposefully place yourself beside excellent programmers.

That is, if you want to be an exceptional programmer then you must consciously place yourself daily amongst people who are exceptional programmers. It's a really simple but profound way to make sure that you improve your skills and attitudes.

We are products of our environment, after all. Just as plants need good soil, fertiliser, and the correct atmosphere to grow healthily, so do we.

Spending too long with depressing people will make you feel depressed. Spending too long with run-down people will make you feel tired and lethargic. Spending too long with sloppy workmen will encourage you to work sloppily yourself – why bother trying if no one else is? Conversely, working with passionate individuals who strive to make better software will encourage you to do the same.

By immersing yourself in the environment of excellent programmers you will treat yourself to:

- Enthusiasm that is infectious
- Motivation that is inspirational
- Responsibility that is contagious

Find people like that and marinate yourself in their company. Consciously seek out the people who care about good code, and about writing it well. In that kind of environment, you won't fail to be nurtured and encouraged.

By working with high calibre developers you will gain far more than technical knowledge; although that knowledge itself is very valuable. You'll enjoy positive reinforcement of good programming habits and attitudes. You'll be encouraged to grow, and be challenged to improve in your weaker areas. This isn't always comfortable or easy, but it *is* worthwhile.

So make a point of seeking out the best programmers and work with them. Design code with them. Pair program with them. Socialise with them.

What to do

You could make this kind of relationship formal with *mentorship*; indeed many good workplaces try to put mentorship schemes into practice formally. Carve out specific chunks of time to work together.

Or you may pursue it informally: get yourself assigned on the same projects as great programmers. Move company to work with them. Go to conferences, talks, or user groups to meet with them. Or just make a point of hanging out with other great programmers.

As you do this, learn from them. Watch out for:

- how they think and solve puzzles,
- how they plan a route into problems,
- the attitude they adopt when things get hard,
- how they know to keep pushing on a problem, when to take a break, or when try a different approach, and
- their specific coding skills and techniques that you don't yet understand.

Know your experts

Consider carefully what you think an excellent programmer looks like.

You specifically *don't* want to get alongside people who work too hard, filling all the hours God sends with code. Those people are almost certainly not the exceptional programmers! Managers often think that employees who spend every waking hour on a project are the programming heroes, but often this really hints at their lack of ability. They can't get things right first time, so they have to spend many more hours getting the code to 'work' than was actually necessary.

Experts make it look easy and get things done on time.

20/20 hindsight

As I look back over my career, I realise that the most enjoyable and personally productive times I've encountered have been when I've been working alongside excellent, motivated, interesting developers. And because of this, I will now always attempt to place myself alongside people like that. I've learnt that they make me better at what I do, and I have more fun whilst I'm doing it.

An interesting and beneficial side-effect of working with good coders is that you are far more likely to end up working with good code. ■

Questions

So ask yourself now:

1. Are you around people you think are excellent programmers right now? Why? Or why not?
2. How can you move yourself nearer better coders? Can you move projects or teams? Is it time to move company?
3. How can you tell who is an excellent developer, and who isn't?

Identify some 'coding heroes' you'd like to learn from, and plan a way to work alongside them. Consider asking to be mentored by them.

References

[1] Frederick P. Brooks (1995) *The Mythical Man Month*, Addison Wesley. ISBN: 0201835959

[2] Tom Demarco and Timothy Lister (1999) *Peopleware* Dorset House Publishing. ISBN: 0932633439

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

The Meaning of Words – Part 9: Libraries, Linkers and Loaders

Francis Glassborow explores another set of computing terms.

I started out to write about linkers and loaders (in response to a reader request). Then I realised that libraries were also part of this and completed a nice trio of ‘L’ words. When I researched ‘program libraries’ I was surprised to find that the concept predates computers. The idea of a library of routines goes back to the days of Jacquard looms that were controlled by punch cards. Cards for particular sub-patterns were kept and could be assembled (linked) to make more complex woven patterns before being loaded into the looms control system.

Charles Babbage picked up the idea of using punch cards to store programs and proposed their use for his Analytical Engine. For many reasons, mostly political and economic, Babbage’s designs were never built though parts of Difference Engines 1 & 2 and a small part of the Analytical Engine were made. We now know that his ideas and designs could have been completed with the technology of the 19th century as two complete working Difference Engine 2s were built over the final decades of the 20th century. Care was taken to ensure that they were manufactured to tolerances that could have been achieved at the time of the original design. We had to wait over 100 years from the designs of Difference Engine 2 and Analytical Engine until working computers were designed and built in the middle of the 20th century.

the idea of delaying some linking till execution time was developed

From the earliest days of working computers, the idea of libraries of routines that could be linked into a programmer’s code was alive and well. In the early days, these libraries had physical form such as punched cards or paper tape before they moved onto magnetic media (initially magnetic tape but then to some form of disc storage). In those pioneering days, all kinds of physical devices were used for memory. To some extent, I think one of the more terrifying was the large magnetic drum memory. Here again, we have surprises. Magnetic drum memory was the first magnetic data storage. This data storage memory device was designed and developed by the Austrian engineer Gustav Tauschek in 1932. Note that date because it means that magnetic storage memory substantially predates working computers. What is often missed is the amount of kinetic energy that was stored in the rotating drum. If one had broken loose from its bearings, it could have done a considerable amount of damage. As an aside, note that high speed rotating devices have been used for several decades for energy storage (an early example was using a flywheel for regenerative breaking on buses).

The reason that long term storage was dominated by paper tape and punched cards is simply a matter of cost. They are also much more durable, but suffer in modern terms from a very low data density. It seems that with each new phase of data storage – from clay tablets to modern cloud storage – we have increased the density of data storage at the cost of durability. I suspect that Babylonian clay tablets will still be readable 2000 years in our future. That is not going to be true of modern data storage devices. It is already becoming difficult to read data from 5.25 inch floppy disks and almost impossible to read data from the previous 7 inch floppy disks.

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

So, back to libraries. These days, a program library is a collection of routines (functions etc.) usually with a common theme from which individual parts can be incorporated into the program you write. You need some way to tell your compiler about the available contents. Languages such as C and C++ do this via header files. Other languages use different mechanisms, but the essential characteristic of a library is that it contains the definitions of many elements and only some of those will be needed in any given program.

We also break our programs up into separate units so that we do not have to deal with million-line monoliths. Somehow, these various bits have to be combined and that is the job of a linker.

When a compiler converts your source code into object code, it includes a list of function calls etc. that have no matching definition. When a linker combines several files of object code it tries to resolve these calls. However, in most programs there will still be some unresolved calls. At this point, the linker will search any libraries that have been provided to attempt to resolve these calls. Sometimes these added definitions from a library themselves include calls to functions that are not defined in the current conglomerate. Single pass linkers (much more common in the past) give a link time failure if there are any unresolved calls after it has searched all the provided libraries. More powerful linkers repeat the search until either all calls are resolved or no further resolutions have been achieved in the latest iteration over the libraries.

If the linker is successful, it generates an executable.

However, some library routines are large and commonly used. Some time in the 1980s, the idea of delaying some linking till execution time (called dynamic linking) was developed so that libraries could be shared across multiple programs running simultaneously. I think we should note that in environments where only a single program is being executed, shared libraries are more resource-expensive than statically linked ones. You may be surprised to learn that most computers run only a single program. Think of all the myriad devices in our world that use dedicated computers. Your modern car has dozens of such devices. Static linking for such devices is essential.

If you would like a more comprehensive summary of linkers, follow this link (☺):

> https://web.stanford.edu/~ouster/cgi-bin/cs140-winter13/lecture.php?topic=linkers

So, your compiler and linker have created an executable but that program has to get from its long-term storage into working storage. This is the job of a loader.

I can recall using a mini-computer in 1982 where I had to manually load the start-up program by using a set of switches on the console. I had to set the 16 switches and press a button to load that instruction into working memory and repeat the process a few times until a very primitive program was complete. That program had a single purpose: it loaded a more substantial operating program from an early hard drive (called a Winchester) that provided basic input and output from a teletype machine and supported access to the hard drive to load whatever program it was that I wanted to use.

Fluent C

Christopher Preschern describes a pattern for managing platform-specific code in C.

C is widespread, in particular with systems where high-performance or hardware-near programming is required. With hardware-near programming comes the necessity of dealing with hardware variants. Aside from hardware variants, some systems support multiple operating systems or cope with multiple product variants in the code. A commonly used approach to addressing these issues is to use `#ifdef` statements of the C preprocessor to distinguish variants in the code. The C preprocessor comes with this power, but with this power also comes the responsibility to use it in a well-structured way.

However, that is where the weakness of the C preprocessor with its `#ifdef` statements shows up. The C preprocessor does not support any methods to enforce rules regarding its usage. That is a pity, because it can very easily be abused. It is very easy to add another hardware variant or another optional feature anywhere in the code by adding yet another `#ifdef`.

Using `#ifdef` statements in such an unstructured and ad hoc way is the certain path to hell. The code becomes unreadable and unmaintainable. This article presents approaches to escape from that `#ifdef` hell and is based on the book *Fluent C* [1] that shows this and other guidance in the form of C programming design patterns that were elaborated at pattern conferences [2].

Isolated primitives pattern

Context

Your code calls platform-specific functions. You have different pieces of code for different platforms, and you distinguish between the code variants with `#ifdef` statements. You cannot simply avoid variants because there are no standardized functions available that provide the feature you need in a uniform way on all your platforms.

Example

Let’s say you want to create a directory inside the home directory of the current user and you want that code to work on Windows as well as on Linux systems. Listing 1 (overleaf) is your first try for that using `#ifdef` statements.

This code is not beautiful. The program logic is completely duplicated. This is not operating system-independent code; instead, it is only two different operating system-specific implementations sequentially put into one function.

Problem

Having code variants organized with `#ifdef` statements makes the code unreadable. It is very difficult to follow the program flow, because it is implemented multiple times for multiple platforms.

When trying to understand the code, you usually focus on only one platform, but the `#ifdef`s force you to jump between the lines in the code to find the code variant you are interested in.

The `#ifdef` statements also make the code difficult to maintain. Such statements invite programmers to only fix the code for the one platform they are interested in and to not touch any other code because of the

CHRISTOPHER PRESCHERN

Christopher is a leading member of the design pattern community and organizes design pattern conferences and initiatives to improve pattern writing. As a C programmer at the company ABB, he gathered and documented hands-on knowledge on how to write industrial-strength code. He has lectured on coding and quality at Graz University of Technology and holds a PhD in computer science

Meaning of Words – Part 9 (continued)

As multi-purpose computers have become ever more complicated, the job of a loader has become more complex in execution but loaders are still just programs to place your executable into working memory and then hand over to your program to do whatever it is designed to do.

Just a final anecdote before I hand over to you to write and tell me about any misconceptions I have revealed in the above.

Back in about 1984, I had a Sinclair Spectrum that was attached to a floppy disk drive. I had written a version of Forth (it was called L-Forth and was used by my students as it ran on both a Sinclair 48K Spectrum and a Research Machines 380Z) that ran in the top 32K of the 48K Spectrum’s RAM. That left the bottom 32K for the 16K ROM along with 16K of RAM which could be used, among other things, to run small Basic programs (and I had added a facility so that L-Forth running on a Spectrum could call out to Basic running in that 16K – more useful than you might think.)

My problem was that the minimal disk system wanted to use 4K bytes and not just any 4K but specifically that located 40K into the RAM. That was too large a proportion of the RAM I had available. I solved this by writing a 128 byte position independent code fragment that swapped the contents of the 4K block to disk and replaced it with the disk system. When finished with the disk operations, the same PIC fragment reversed the procedure.

Eventually I developed a suite of programs that could:

- transfer data from tape to a disk
- inspect the code of a game for 128 bytes of reusable memory (games would have some working storage that I could borrow)
- insert the PIC fragment in that space
- insert the disk system into the game copying the replaced 4K to a fixed location on the disk

Now I had a bootstrap loader.

1. Use the disk system to load the modified game code.
2. Use the PIC fragment within the game code to restore the 4K borrowed for the disk system from the fixed location on the disk
3. Start the game.

Those days were a lot of fun as well as being very instructive.

As always, comments are very welcome and corrections more so. I have always followed the principle that it is a condition of my sharing my ideas and understanding that readers correct them as appropriate. ■

Listing 1

```
#include <stdio.h>
#include <stdlib.h>
#ifdef __unix__
#include <sys/stat.h>
#elif defined _WIN32
#include <windows.h>
#endif

int main()
{
  char dirname[50];
  #ifdef __unix__
  sprintf(dirname, "%s%s", getenv("HOME"),
   "/newdir/");
  #elif defined _WIN32
  sprintf(dirname, "%s%s%s", getenv("HOMEDRIVE"),
    getenv("HOMEPATH"), "\\newdir\\");
  #endif
  #ifdef __unix__
  mkdir(dirname,S_IRWXU);
  #elif defined _WIN32
  CreateDirectory(dirname, NULL);
  #endif
  return 0;
}
```

danger of breaking it. But only fixing a bug or introducing a new feature for one platform means that the behaviour of the code on the other platforms drifts apart. The alternative – to fix such a bug on all platforms in different ways – requires testing the code on all platforms.

Testing code with many code variants is difficult. Each new kind of `#ifdef` statement doubles the testing effort because all possible combinations have to be tested. Even worse, each such statement doubles the number of binaries that can be built and have to be tested. That brings in a logistic problem because build times increase and the number of binaries provided to the test department and to the customer increases.

Solution

Isolate your code variants. In your implementation file, put the code handling the variants into separate functions and call these functions from your main program logic, which then only contains platform-independent code.

Each of your functions should either only contain program logic or only cope with handling variants. None of your functions should do both. So either there is no `#ifdef` statement at all in a function, or there are `#ifdef` statements with a single variant-dependent function call per `#ifdef` branch. Such a variant could be a software feature that is turned on or off by a build configuration, or it could be a platform variant as shown in the code in Listing 2.

Utilizing a single function call per `#ifdef` branch should make it possible to find a good abstraction granularity for the functions handling the variants. Usually the granularity is exactly at the level of the available platform-specific or feature-specific functions to be wrapped.

Example resolved

Let's apply that solution to the example that creates a directory (Listing 3).

Consequences

The main program logic is now easy to follow, because the code variants are separated from it. When reading the main code, it is no longer necessary to jump between the lines to find out what the code does on one specific platform.

To determine what the code does on one specific platform, you have to look at the called function that implements this variant. Having that code in a separately called function has the advantage that it can be called from other places in the file, and thus code duplications can be avoided. If the functionality is also required in other implementation files, then the Abstraction Layer Pattern has to be implemented.

No program logic should be introduced in the functions handling the variants, so it is easier to pinpoint bugs that do not occur on all platforms, because it is easy to identify the places in the code where the behavior of the platforms differs.

Listing 2

```
void handlePlatformVariants()
{
  #ifdef PLATFORM_A
    /* call function of platform A */
  #elif defined PLATFORM_B
    /* call function of platform B */
  #endif
}

int main()
{
  /* program logic goes here */
  handlePlatformVariants();
  /* program logic continues */
}
```

Code duplication becomes less of an issue since the main program logic is well separated from the variant implementations. There is no temptation to duplicate the program logic anymore, so there is no threat of then accidentally only making bug fixes in one of these duplications.

What's next?

In this article, you saw one of many steps how to improve the platform-specific example code for creating directories. Further steps would be to put platform-specific code below an abstraction layer and have platform variants on a code-file level to use tools like Make for distinguishing between the variants. All these steps as well as patterns about other topics relevant for C programmers, like error handling or API design, are covered in the O'Reilly book, *Fluent C* [1]. With these patterns and examples for their application, the book helps you further your C programming knowledge to a level that allows you to write robust and industrial-strength code. ■

References

[1] Christopher Preschern (2022) *Fluent C*, published by O'Reilly (see https://www.oreilly.com/library/view/fluent-c/9781492097273/), ISBN 9781492097334

[2] Christopher Preschern 'Patterns to escape the #ifdef hell', *EuroPLop '19: Proceedings of the 24th European Conference on Pattern Languages of Programs*, July 2019, published on the ACM Digital Library and accessible from https://dl.acm.org/doi/10.1145/3361149.3361151

Listing 3

```
void getDirectoryName(char* dirname)
{
  #ifdef __unix__
    sprintf(dirname, "%s%s", getenv("HOME"),
      "/newdir/");
  #elif defined _WIN32
    sprintf(dirname, "%s%s%s",
      getenv("HOMEDRIVE"), getenv("HOMEPATH"),
      "\\newdir\\");
  #endif
}
void createNewDirectory(char* dirname)
{
  #ifdef __unix__
    mkdir(dirname,S_IRWXU);
  #elif defined _WIN32
    CreateDirectory (dirname, NULL);
  #endif
}
int main()
{
  char dirname[50];
  getDirectoryName(dirname);
  createNewDirectory(dirname);
  return 0;
}
```

Reporting An MSvc Runtime Library Bug

Roger Orr shares his experience of submitting a bug fix to Microsoft.

In code critique #138 the example code attempted to access an iterator after it had been invalidated by erasing an earlier item in the vector. The code critique problem description said: “But when I compile with msvc in debug mode it aborts”. While the main ‘point’ of the code critique was the access of an invalid iterator, the reason for the abort in MSVC is a ‘missing’ debug mode check in **`<vector>`** the MSVC runtime library.

This made the code critique slightly more challenging, as the abort doesn’t immediately give away the solution to the problem.

However, I thought that a public-spirited action would be to report the problem to Microsoft (but only after the CC deadline had passed) and originally I had intended to simply report the bug. But things have changed recently with MSVC’s runtime library – the source code is hosted publicly on github and so you can report issues and also raise P/Rs (pull requests i.e.requests to merge your work into the main branch).

While doing this I thought it might be instructive to turn it into a *CVu* article in case other people come across similar problems and are put off reporting them.

The git hub repository

The Microsoft STL is hosted at https://github.com/microsoft/STL. The source code for the problem uncovered by the code critique at the time the Code Crique was written is visible at https://github.com/microsoft/STL/blob/2f03bdf/stl/inc/vector#L45-L54 and looked like Listing 1.

Listing 1

```
45. _NODISCARD _CONSTEXPR20 reference operator*() const noexcept {
46.   #if _ITERATOR_DEBUG_LEVEL != 0
47.     const auto _Mycont = static_cast<const _Myvec*>(this->_Getcont());
48.     _STL_VERIFY(_Ptr, "can't dereference value-initialized vector iterator");
49.     _STL_VERIFY(
50.       _Mycont->_Myfirst <= _Ptr && _Ptr < _Mycont->_Mylast, "can't dereference out of range vector
iterator");
51.   #endif // _ITERATOR_DEBUG_LEVEL != 0
52.
53.   return *_Ptr;
54. }
```

The bug was that **`this->_Getcont()`** returned null after the vector erase method is called, so dereferencing **`_Mycont`** in line 49/50 caused an access violation.

Analysis

The Microsoft compiler defaults **`_ITERATOR_DEBUG_LEVEL`** to 0 in a ‘release’ build and to 2 in a ‘debug’ build.

Looking at the code in this header, and in **`<xmemory>`** where the iterator base class is defined, reveals that the base class for the iterator in a debug build (**`_Iterator_base12`**) contains a field **`_Myproxy`** which is set for a valid iterator and set to null when **`_ITERATOR_DEBUG_LEVEL`** is 2 in a private member function of vector, **`_Orphan_range_unlocked()`** (which at this point was at line 2217):

```
2216.             } else { // orphan the iterator
2217.                 _Temp->_Myproxy = nullptr;
```

Has someone already reported – or even fixed – this issue?

The first check is that no-one has already reported the issue – raising a duplicate bug report simply causes extra work for other people. I did a quick search verious terms in the open issues and their titles and didn’t find any that appeared to be relevant.

It doesn’t seem to be a hard problem to trigger, but since the debugging iterator does give an error, via the resultant access violation, perhaps this gives enough information to others hitting this issue that there was insufficient motivation to try and resolve it.

Proposed resolution

The obvious fix here seems to be to add a check for **`_Mycont`** not being **`null`**, just before it is dereferenced.

It must be checked *after* **`_Ptr`** is checked, as we don’t want to mask the problem when a value-initialised iterator is dereferenced, and for such an interator **`_Mycont`** will be also initialised to **`null`** in a debug build. See Listing 2.

But wait … there’s more

We are not done, however – whenever you find a bug it is almost always worth checking to see if the same bug is repeated elsewhere. In this case, there are three *other* methods in this class (**`_Vector_const_iterator`**) that perform a similar check: **`operator->()`**, **`operator++()`**, and **`operator--()`**. There is also a method in a *related* class, **`_Vector_iterator`**, which performs the same check. (I missed this one in my first analysis!)

There is also a similar, but not identical, check in an internal method, **`_Verify_offset`**, which I didn’t cover.

Checking the code

I first of all forked the microsoft STL repository in github, using the Fork button in the UI, to my own github account (https://github.com/rogerorr/STL/) and then cloned this locally with the command:

```
git clone git@github.com:rogerorr/STL.git
--recurse-submodules
```

The `README.md` file in the STL project contains clear instructions for building and running the tests for the runtime library. I already had a suitable versions of cmake, ninja, and python on my laptop, so there was in this case no need for me to fetch other programs before I began.

Following the instructions, I opened a command window using the ‘x64 Native Tools Command Prompt for VS 2022’ shortcut, changed to the STL directory I had just downloaded and ran:

```
cmake -G Ninja -S . -B out\build\x64
ninja -C out\build\x64
```

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.co.uk

Listing 2

```
_NODISCARD _CONSTEXPR20 reference operator*() const noexcept {
  #if _ITERATOR_DEBUG_LEVEL != 0
    const auto _Mycont = static_cast<const _Myvec*>(this->_Getcont());
    _STL_VERIFY(_Ptr, "can't dereference value-initialized vector iterator");
    _STL_VERIFY(_Mycont, "can't dereference invalidated vector iterator");
    _STL_VERIFY(
      _Mycont->_Myfirst <= _Ptr && _Ptr < _Mycont->_Mylast, "can't dereference out of range vector iterator");
  #endif // _ITERATOR_DEBUG_LEVEL != 0

  return *_Ptr;
}
```

I then ran some of the tests (by executing **ctest** in the `out\build\x64` subdirectory) to 'kick the tyres' for the build.

Now I created a branch in my local workspace, made my proposed changes to the **<vector>** header in `stl\inc` and confirmed the tests still passed. This took a long time – I went to bed leaving my machine running – and the results were:

```
Skipped          :   397
Unsupported      :  2967
Passed           : 28716
Expectedly Failed:   930
Failed           :    30
```

I was initially concerned, but on closer investigation the failed were all tests of locale, so presumably were being caused by something about my machine setup (or lack of it) – they certainly appeared to have nothing to do with the changes to **<vector>** so I ignored these failures for now!

Did it work?

Of course, passing the tests only proved I'd not broken anything obvious, not that the original problem was resolved. The original problem was in a code critique, but it was a simple matter to extract a very small test case that demonstrates the issue. In general, though, it can be quite a challenge to simplify a program that demonstrates a bug to a *minimal* program that still demonstrates the same bug.

```
#include <vector>
int main() {
  std::vector<int> v({0,1,2,3});
  auto iter = v.begin();
  v.erase(v.begin()); // invalidate iter
  return *iter;       // access violation, rather
                      // than an assertion
}
```

In this case, given what was changed, it would be enough to recompile my test case with the modified **<vector>** header to check that the expected assert now triggers. In more complicated cases, you'd need to link against a changed library, too.

MS generate a batch script (`set_environment.bat)` in the build output directory that you can run to insert the proper paths for the built headers and libraries in the **INCLUDE**/**LIB**/**PATH** environment variables.

In this simple case, testing with the modified **<vector>** can be done from the Visual Studio command prompt just by using:

```
cl /I c:\projects\GitHub\rogerorr\STL\stl\inc
/EHsc /MDd /Zi repro.cpp
```

When executed, we get the expected assertion, rather than an abort (see Figure 1).

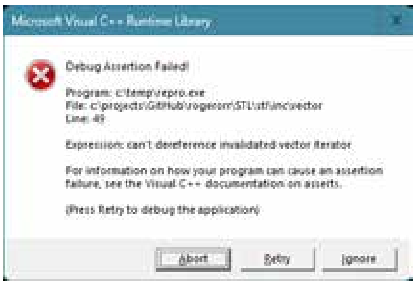


Figure 1

Microsoft Visual C++ Runtime Library

Debug Assertion Failed!

Program: c:\temp\repro.exe
File: c:\projects\GitHub\rogerorr\STL\stl\inc\vector
Line: 49

Expression: can't dereference invalidated vector iterator

For information on how your program can cause an assertion failure, see the Visual C++ documentation on asserts.

(Press Retry to debug the application)

Raising a P/R

We now have everything we need to report the problem and raise a P/R. First off I raised an issue, including the simple reproduction of the problem and my analysis: https://github.com/microsoft/STL/issues/3281 This was straightforward.

I then raised a P/R in guthub from my branch to the Microsoft main branch. The documentation says that the Microsoft Contributor License Agreement bot will automatically request completion of the appropriate document for a new contributor; this check passed automatically, which I originally assumed might be because a prior NDA (Non Disclosure Agreement) I have signed with MS associated with the same email account, but subsequent comments indicate that the bot doesn't trigger for 'sufficiently small' changes.

The repository on github has been set up to run a range of tests on the committed code automaticallly, giving improved confidence that the change is not going to break something I failed to test!

Figure 2 shows a subset of the check output from the github page https://github.com/microsoft/STL/pull/3282

Figure 2

microsoft.STL (Build and Test x86 6) Successful in 7m — Build and Test x86 6 succeeded
microsoft.STL (Build and Test x86 7) Successful in 10m — Build and Test x86 7 succeeded
microsoft.STL (Build and Test x86 8) Successful in 7m — Build and Test x86 8 succeeded
microsoft.STL (Build arm 1) Successful in 12m — Build arm 1 succeeded
microsoft.STL (Build arm 2) Successful in 5m — Build arm 2 succeeded
microsoft.STL (Build arm 3) Successful in 11m — Build arm 3 succeeded

A few comments were raised, and answered, and the P/R was approved by three people and was ready to be merged. STL merged the work a day or two later, and added the comment that the change will be in 'VS 2022 17.6 Preview 1'.

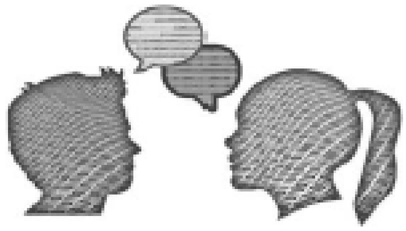
It is quite gratifying to feel I have made a small improvement to a compiler I use a lot.

Conclusion

I was very pleased by the way the process worked and how responsive the MS engineers were – and I thank Nicole Mazzuca, Casey Carter, and Stephan T. Lavavej for their rapid approvals; so I am encouraged that getting bugs reported and fixed in the Microsoft STL is now much more like the existing processes for gcc and clang. The advantages for the user include transparency of the open issues, the ability to provide a P/R (and to test it yourself), and the ability to achieve such a quick turn around. ■

Code Critique Competition 139

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



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

Last issue's code

> I'm puzzled by the behaviour of the vector iterator in the following program (Listing 1). When I compile with gcc and msvc in release mode I get what I expect, for example:
>
> ```
> *it1: 5 ptr: 000001F29EAFCDF8
> *it2: 6 ptr: 000001F29EAFCDF8
> *ptr: 6
> *it1: 6
> it1 == it2? true
> ```
>
> But when I compile with msvc in debug mode it aborts:
>
> ```
> *it1: 5 ptr: 000001D6DE61F7A8
> *it2: 6 ptr: 000001D6DE61F7A8
> *ptr: 6
> *it1:
> ```
>
> I was expecting that the iterator would behave like the pointer and since what is that address has changed from 5 to 6, it'd simply reference 6. I don't understand how the behaviour of the `const` iterator object `it1` can have changed – I've not gone off the end of the vector. (See Listing 1.)

Listing 1

```
#include <iostream>
#include <vector>
int main() {
  std::vector<size_t> v;
  for (size_t i = 0; i != 10; ++i) {
    v.push_back(i);
  }
  // v = {0,1,2,3,4,5,6,7,8,9}

  const auto it1 = v.begin() + 5;
  const auto *ptr = it1.operator->();
  // it1 => 5

  std::cout << "*it1: " << *it1
            << " ptr: " << ptr << '\n';
  v.erase(v.begin() + 2);
  // v = {0,1,2,3,4,6,7,8,9}

  const auto it2 = v.begin() + 5;
  // it2 => 6
  std::cout << "*it2: " << *it2 << " ptr: "
            << it2.operator->() << '\n';
  try {
    // ptr now points to 6 not 5
    std::cout << "*ptr: " << *ptr << '\n';
    // I expect it1 to also point to 6
    std::cout << "*it1: " << *it1 << '\n';
    // And I expect it1 to equal it2
    std::cout << "it1 == it2? "
              << (it1 == it2 ? "true"
                             : "false")
              << '\n';
  } catch (std::exception ex) {
    // Never get here :(
    std::cerr << "Exception: " << ex.what()
              << '\n';
  }
}
```

Critiques

Paul Floyd < pjfloyd@wanadoo.fr>

It's a long time since I used the MS compiler, though I do usually have it installed just in case. This CC seems to be one of those cases.

First off, I tried this with clang++ and GCC, running with Valgrind and Sanitizers. Not a peep. Then I tried with Visual Studio and reproduced the problem. cppreference explains why there is a problem [1] and in particular:

> Invalidates iterators and references at or after the point of the erase, including the `end()` iterator.

That means that erasing the third element will make an iterator to the sixth element invalid.

If that's the case then why is there no problem with clang++ or g++? Getting out my PDF of ISO/IEC 14882-2012 (the C++11 standard) I see that 24.2.1 para 10 says:

> An invalid iterator in an iterator that may by singular.

This then refers to footnote 267 which says:

> This definition applies to pointers, since pointers are iterators. The effect if dereferencing an iterator that has been invalidated is undefined.

My hand-waving expectation for what `std::vector::erase` does is

1. A memmove of the part of the vector that follows the erased element(s) downwards so that they overwrite the erased element(s). In this case, erasing the third element means that the 4th to 10th get moved down to become the 3rd to 9th.
2. The size of the vector gets reduced by the number of elements erased. In this case it's the single iterator position overload of erase that's being used, so one element is erased and the size gets decreased from 10 to 9.

Continuing to gesticulate, I also expect that an `std::vector::iterator` is roughly synonymous with a pointer. g++, clang++ and Visual Studio optimized builds seem to be doing all of the above, so after the erase `it1` is still pointing to the same memory but the contents have been shifted down so the contents of that memory will have changed from 5 to 6.

One minor thing at this point, the comment here

```
v.erase(v.begin() + 2);
// v = {0,1,2,3,4,6,7,8,9}
```

is wrong, that should be `// v = {0,1,3,4,5,6,7,8,9}` since the 3rd element (value 2) was erased, not the sixth one.

Getting back to what is going on in Visual Studio. With a bit of debugging I see that `_Orphan_range(_Whereptr, _Mylast);` is being called. This seems to clear all iterators that point to that range. The

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.co.uk

implementation code for this isn't that straightforward, but it looks like it is using a **`_Vector_val`** for vector elements which has a **`_Container_proxy*`** which, amongst other things, contains a linked list of iterators. So when the code tries to dereference **`it1`** a debug only check tries to get the container from the iterator. This container was cleared and **`nullptr`** gets returned.

The **`std::vector`** code in question starts on line 45 of the `vector` header file in VS 2022 17.3.6. It's Apache 2.0 licensed, so I'll copy it here (ED: slightly reformatted for publication).

```
_NODISCARD _CONSTEXPR20 reference
operator*() const noexcept {
  #if _ITERATOR_DEBUG_LEVEL != 0
  const auto _Mycont =
    static_cast<const _Myvec*>(this->_Getcont());
  _STL_VERIFY(_Ptr, "can't dereference "
    "value-initialized vector iterator");
  _STL_VERIFY(
    _Mycont->_Myfirst <=
      _Ptr && _Ptr < _Mycont->_Mylast,
      "can't dereference out of range vector "
      "iterator");
  #endif // _ITERATOR_DEBUG_LEVEL != 0
  return *_Ptr;
}
```

It's a good thing that MS has added this extra checking code. Though I do think that they could perhaps improve it a bit. At the moment, it checks that the pointer is valid and that the pointer points to a valid element within the range of the vector. They could add something like

```
_STL_VERIFY(_Mycont,
"can't dereference invalidated vector iterator");
```

(which would be between the two existing **`_STL_VERIFY`**s above).

The use of **`ptr`** circumvents this checking.

In order to 'fix' this problem:

1. Make **`it1`** non-const
2. Re-assign **`it1`** after the erase:

```
v.erase(v.begin() + 2);
it1 = v.begin() + 5;
```

Last thing, I did try the VS code analysis and it didn't complain. Maybe some tools will be able to detect this (since the vector changes are 'static' in the sense that a literal is used for the element to erase, and all of the code is local to **`main`**).

Reference

[1] CPP Reference : https://en.cppreference.com/w/cpp/container/vector/erase)

James Holland <jim.robert.holland@gmail.com>

The comment after the erase statement does not agree with the statement. Assuming the code is correct, the comment should be

```
// v = {0,1,3,4,5,6,7,8,9}
```

not

```
// v = {0,1,2,3,4,6,7,8,9}
```

The student's program first creates a vector of numbers between 0 and 9 inclusive. Then the iterator, **`it1`**, is initialised with a value that points to the fifth element of the vector. The underlying pointer of the iterator is then used to initialise a pointer. We now have an iterator and a pointer both referring to the fifth element of the vector. All this is perfectly fine. The iterator and the pointer are valid and can be used to obtain the value of the vector's fifth element.

The program then erases element 2 of the vector. The **`erase`** function does this by shifting all the elements of the vector beyond element 2 down one position and reducing the size of the vector by one. Anything that was pointing to an object at element 2 now points to a different object. Because of this, such iterators and pointers are considered invalid after the erase operation.

The C++ standard states that **`erase`**:

> Invalidates iterators and references at or after the point of the erase.

When in debug mode, MSVC adds code to a program with the aim of detecting undefined behaviour including the use of invalid iterators. This is why the program aborts when in debug mode, presumably with some diagnostic message, as the use of invalid iterators has been detected. When in release mode, no such code is added and the program produces the expected result. By default, gcc does not provide the same level of protection and so produces the expected result irrespective of the mode it is in. To activate gcc iterator checking, define **`_GLIBCXX_DEBUG`**. Apparently, MSVC and gcc do not provide similar checks for raw pointers or references. Perhaps this is more difficult or impossible in the general case.

It could be that a pointer or iterator at or beyond the erased element still refers to a valid object and that no harm will come of reading the value of this new object by means of the pointer or iterator. It could be argued that the standard is particularly harsh in this respect. Nevertheless, as the standard states that such iterators and pointers are invalid, dereferencing them results in undefined behaviour.

One way to prevent entering the realms of undefined behaviour is to use indexes as opposed to iterators. The troublesome statement could be replaced by something like the following.

```
std::cout << "v[5]: " << v[5] << '\n';
```

The programmer must still ensure that the index is within the range of the vector in the general case, however.

Another way to solve the problem is to reevaluate the iterator, **`it1`**, after the erasure. The student has done this with **`it2`** just after the **`erase`** statement.

The use of **`it.operator->()`** is interesting. It returns a pointer to the object to which the iterator is pointing. This is what is required but perhaps **`&*it`** would be simpler. Both forms generate exactly the same machine code and, therefore, have the same effect.

It would appear that the student is attempting to use a try-catch block to detect the use of invalid iterators and pointers. Unfortunately, this approach will not work. By default, operator **`std::operator<<()`** does not throw and even when configured to throw on errors an invalid iterator or pointer will not be recognised as an error. Comparing for equality invalid iterators will not throw an exception either.

The best recourse the student has is to compile the program in debug mode and to define **`_GLIBCXX_DEBUG`** if using gcc. Any compiler errors and warnings should be addressed. Once the program compiles without warnings, it can be run and tested. Any run-time diagnostic messages will indicate undefined behaviour and can be corrected. After thorough testing and the correction of all run-time deficiencies, the program can be compiled in release mode (and with **`_GLIBCXX_DEBUG`** not defined) if desired.

Incidentally, the iterator **`it1`** is declared as const. This means that it cannot be modified to point to another object. This is consistent with the behaviour of the student's program. Additionally, the iterator is never used to change the value of the object it points to. This feature can be expressed in the code by declaring a **`const_iterator`** as opposed to an **`iterator`**. This can be easily achieved, when using **`auto`**, by employing **`cbegin()`** rather than **`begin()`**.

As well as iterators, pointers can also be declared so that they can point only to their initialised object. This is achieved by the use of **`const`** to the right of the asterisk in the declaration of the pointer. In the case of the student's code, the declaration (and initialisation) becomes **`const auto * const ptr = it1.operator->()`**. Note that having initialised **`it1`** with **`v.cbegin()`**, the **`const`** to the left of the asterisk in the declaration of `ptr` becomes redundant. If the iterator cannot change its pointed-to object, then neither can the pointer. The use of the redundant `const` does no harm and may even enhance readability.

Instead of writing an explicit `for`-loop to set the elements of vector, `v`, the following statements are preferred as, almost certainly, more efficient machine code will be generated. Things will be even simpler when the range-based `iota` is introduced in C++23.

```
std::vector<size_t> v(10);
std::iota(v.begin(), v.end(), 0);
```

Finally, exceptions should be caught by `const` reference. Therefore, the statement in the student's program should be:

```
catch (const std::exception & ex)
```

Commentary

Visual Studio enables additional checking of standard library objects when a debugging build is selected. Other compilers also have similar functionality, but it may not be selected automatically. One reason for this is that tracking the state of containers and iterators can rapidly get very expensive and render such a build almost unusable; deciding which checks you should get 'by default' is a design choice between various competing forces.

For gcc this checking is not automatically selected, but you can enable it by defining the pre-processor symbol `_GLIBCXX_DEBUG`, as James explained above. The output that results for me looks like this:

```
*it1: 5 ptr: 0x7fffcd050f58
*it2: 6 ptr: 0x7fffcd050f58
*ptr: 6
/usr/include/c++/7/debug/safe_iterator.h:270:
Error: attempt to dereference a singular iterator.

Objects involved in the operation:
  iterator "this" @ 0x0x7fffd56e70a0 {
    type = __gnu_debug::_Safe_iterator ...
    state = singular;
    references sequence with type
      'std::__debug::vector<unsigned long,
       std::allocator<unsigned long> >'
      @ 0x0x7fffd56e7190
  }
Aborted (core dumped)
```

The intent for MSVC is to get similar output, but it so happens that (as Paul explained) the debugging code itself is missing a check and so the program simply aborts. However, if you are in a debugger, the faulting location does give a pretty clear indication of the problem with the code. I considered this a bug in their debugging iterator – as did Paul Floyd – and I have raised a P/R against the `<vector>` header to get this improved in the future. (I didn't raise the P/R until after the deadline for this *CVu*…) I note that Paul's proposed wording exactly matches mine – a good micro-example of 'independent development' not resulting in independent code.

Compiler and library writers have put in a lot of effort to enable such debugging, so knowing how to enable it for your own codebase can detect quite a few problems when you are testing – and if you are able to build your code with multiple compilers this can further increase the testing of your code. Of course, other tools can also be used to help find problems like this one, but those that are supplied as part of the compiler do have certain obvious advantages! Paul's experience with both valgrind and sanitisers also shows that no tool seems to be able to find *everything*.

The code also demonstrates an advantage of using a class – the iterator – over a raw pointer as the behaviour of the class can be customised to provide additional checking much more easily than with a raw object. In general programming at a higher level of abstraction can help tooling as the *intent* of the constructs being used is clearer.

The Winner of CC 138

Both entries identified the root cause of the problem and also provided a way to make the program valid.

They also noticed the incorrect comment in the code. Some comments have a nasty way of 'rotting'. When a comment describes behaviour of the code itself – as here – one option would be to replace the comment with code. In this example, which is a test case, that immediately makes the code self-documenting.

James additionally went into a bit more detail about the various ways to making an iterator (and a pointer) `const`-correct, which was useful. He also pointed out the non-idiomatic `catch` statement which takes a copy of the throw object, (although he did not explain *why* this is better – the original code catches a 'slice' of the thrown exception object, which loses information about the exception.)

On balance I felt that James' critique was the better one, so I have awarded him this issue's prize.

Code Critique 139

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

> I'm trying to implement mean and standard deviation in C but my program is not printing what I expect. I also get different answers with MSVC and GCC.
>
> When running with input 1 2 3 4 5 6 7 8 9 10.
>
> Expected: `5.5 3.02765`
>
> GCC: `5.500000 2.549510`
>
> MSVC: `5.500000 -256.000000`

The code is in Listing 2.

Listing 2

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
  double *ptr = malloc(argc * sizeof(double));
  for (int i = 1; i < argc; i++) {
    sscanf(argv[i], "%lf", ptr + i);
  }
  double mean = 0;
  for (int i = 1; i < argc; i++) {
    mean += ptr[i];
  }
  mean /= argc - 1;
  double variance = 0;
  for (int i = 1; i < argc; i++) {
    variance += pow(ptr[i] - mean, 2);
  }
  double stdev = sqrt(variance / argc - 1);
  printf("Sample mean: %lf, stdev %lf\n",
         mean, stdev);
  return 0;
}
```

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 (https://accu.org/menu-overviews/journals-overview/).

This particularly helps overseas members who typically get the magazine much later than members in the UK and Europe.

Report on Homework Challenge 13

Francis Glassborow reports on the last challenge and sets another.

I have only had two submissions at the time of writing. I suspect this is because I forgot to include a deadline. On average submissions arrive at the deadline plus/minus two days so clearly having a submission date is important.

The exception to the above is Silas whose submission is usually within a couple of days of his receiving the print issue of *CVu*. This time he promptly delivered a very terse solution in Python 3. That choice of language was governed among other things by a desire to use Unicode to represent the 'digits' (not sure that we have a word to refer to the individual elements of a number that is neither in base ten (digit) or base two (bit) – perhaps readers might like to contribute suggestions).

James Holland hit the unspecified but intended deadline with a submission that is close to my basic expectations (my original program for my QL was in Basic ☺).

Let me outline a few ideas taken from my memory of the original enhanced by more modern facilities.

As James noted, the string representing the number grows so we need an expandable container. Though the problem is essentially a property of a string representing a number, actually using a string is suboptimal as it would need to be decomposed into individual characters (digits) to process both reversing and, more importantly, addition. James noted that a C++ vector can be expensive to extend but seems to have missed the facility to reserve space and so avoid expensive relocation when the vector needs to grow. However, I think that vector is the wrong container exactly because of its growth mechanism. C++ provides other standard containers that have better performance when growing. My choice would be a deque (double ended queue) but without benchmarking I cannot estimate how that would perform against a vector with a very large reserve.

The issue raised by Silas is well answered by James. Actual printed representations are only needed for input or output. Providing the numerical values of the individual place rather than encoding these as distinct symbols seems sensible. However, if you want a more traditional representation for input and output (internally numerical representation is clearly the right way forward) then my choice would be a lookup table to map symbols to values.

Now, here is an idea from my original program: do not bother to reverse the string! Assuming that the string has (*n*+1) digits simply process the collection by adding element *x* to element (*n-x*) for values of *x* from zero to (*n*+2)/2 (rounded down) and store the result in both position *x* and position *n-x*. That assumes we are using a zero based position. If you find this confusing, try using a paper and pencil to see how it works.

Note that the above procedure is independent of the base.

Now set a carry flag to zero and do a single pass from the units place upwards checking each entry to see if it is >= the base. If it is, subtract the base and increment the next entry by one (and set the overall carry flag). After this pass check the carry flag. If it is still zero, you are done – the result was a palindrome. Unfortunately, the existence of a carry somewhere does not preclude the result being a palindrome (e.g. 47 + 74 = 121). I suspect that exceptions for longer strings are rare but not non-existent (e.g. 407 + 704 = 1111). (It would be interesting to have the program highlight these when they occur.)

FRANCIS GLASSBOROW

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

We need to check the container by comparing each position *x* with position *n-x* for *x* from zero to (*n*+2)/2 (rounded down). Actually, that will include one superfluous comparison when (*n*+1) is odd. Note that this requires exactly the same transversal as that used in stage 1 So, why not combine them?

This gives our inner processing loop something like (using an old C++ `for` loop):

```
bool palindrome = true;
for bool carry = false;
(i=0; i<= (n+2)/2; i++){
  if (digit[i] != digit[n-i] palindrome = false;
  digit[i] += digit[n-i], digit[n-i] = digit[i];
}
/* and now do the adjustment for base which may
need an extra element for the container. */
```

As James observed, it makes more sense to have a big endian representation with the largest place value at the end.

Enough from me.

From Silas Brown

Here's a brief go at HW13 (Listing 1), fitting in 48 columns for publication. The problem says to use characters for digits up to base 100, but there's only 95 characters in ASCII, so I'm using Python 3 for Unicode. I assume no combining diacritics.

Thanks.   Silas

Listing 1

```
b=int(input("Base to use for the numbers: "))
if b<=36: c="0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"
else: c=input("Characters to use for digits: ")
assert b<=len(c), "Not enough characters provided"
assert len(c)==len(set(list(c))), "Duplicate char"
s=list(input("Starting number: ")) ; i,l=0,list
assert all(x in c for x in s), "Unknown character"
while not s==l(reversed(s)):
  t,u,n,r,z,s=s[:],1,0,l(reversed(s)),c.index,[]
  while t: n,u=n+u*(z(t.pop())+z(r.pop())),u*b
  while n: s,n=[c[n%b]]+s,n//b
  print ("".join(s)) ; i += 1
print (f"Palindrome after {i} iterations")
```

Francis: Any ordinary student who submitted that should definitely fail ☺.

From James Holland

This has been quite a challenge for the student. I am informed that he spent an inordinate amount of time writing and rewriting the code. An enjoyable experience, no doubt. The student has come up with a way of entering a number in just about any integer base. As it stands, the program has no run-time human interface. The range of numbers to test, the number base and the maximum number of iterations have to be entered in the source code and the program compiled and run. The range of numbers to be tested is entered with digits separated by a comma. This allows numbers of any base to be entered. For example, 123456710 is equal to 14,13,4,14,1017 and so can be entered as {14, 13, 4, 14, 10}. The number base is set by assigning the `Number::base` variable.

The digits of a number are stored in a vector, each element being of type int. The length of the vector is equal to the number of digits in the number. When two numbers are added there exists the possibility of a carry making the resulting vector one element longer. Adding the extra element could cause the vector to reallocate memory. This is a relatively time-consuming operation. It was, therefore, decided to create the result vector with one extra element before performing the sum. If there is a carry, the extra element would be used. If not, the extra element would have to be removed. This could cause another problem. If the extra element was at the beginning of the vector, all the other digits would have to be shifted down one position if there was no carry. To alleviate this problem, the extra element is located at the end of the vector. Now, if there is no carry, the last element of the vector can be removed. This is a fast operation as no reallocation occurs. Unfortunately, to have a sensible number layout within the vector requires that the least significant digit has to be at the beginning of the vector and the most significant at the end. This is opposite to how one would normally represent the digits of a number in a vector. To compensate, the order of digits has to be reversed when entered and printed. This scheme seemed a good idea at the time but any advantage has not been verified.

Although the program is not particularly user-friendly, it does provide a basis from which a program more tailored to the user’s requirements can be written (Listing 2).

Listing 2

```
#include <algorithm>
#include <cassert>
#include <iostream>
#include <vector>
class Number{
public:
  Number(std::initializer_list<int> num);
  [[nodiscard]] bool is_palindrome() const;
  Number & operator++();
  void reverse_and_add();
  friend std::ostream & operator<<(
    std::ostream & os, const Number & num);
  friend bool operator<(const Number & lhs,
    const Number & rhs);
  inline static int base = 10;
private:
  std::vector<int> number;
};

Number::Number(const std::initializer_list<int>
num) : number(num){
  // Make sure all digits are less than the base.
  assert(std::ranges::all_of(num, [](const int i)
    {return i < base;}));
  std::ranges::reverse(number);
}
bool operator<(const Number & lhs,
  const Number & rhs)
{
  if (lhs.number.size() < rhs.number.size())
    return true;
  if (lhs.number.size() > rhs.number.size())
    return false;
  return std::lexicographical_compare(
    lhs.number.crbegin(), lhs.number.crend(),
    rhs.number.crbegin(), rhs.number.crend());
}
Number &Number::operator++(){
  auto iterator = number.begin();
  int carry = 1;
  while (carry == 1)  {
    if (iterator == number.end())  {
      number.push_back(1);
      carry = 0;
    }
    else  {
     if (*iterator + 1 == base) {
        *iterator = 0;
        ++iterator;
     }
      else {
```

Listing 2 (Cont’d)

```
        *iterator += 1;
        carry = 0;
      }
    }
  }
  return *this;
}
void Number::reverse_and_add(){
  auto number_iterator = number.cbegin();
  auto reverse_iterator = number.crbegin();
  std::vector<int> result(number.size() + 1, 0);
    // Add an element in case of a carry.
  auto result_iterator = result.begin();
  int carry = 0;
  while (number_iterator != number.cend()) {
    if (const auto sum = *number_iterator +
        *reverse_iterator + carry; sum >= base) {
      *result_iterator = sum - base;
      carry = 1;
    }
    else {
      *result_iterator = sum;
      carry = 0;
    }
    ++number_iterator;
    ++reverse_iterator;
    ++result_iterator;
  }
  if (carry == 1) {
    *result_iterator = 1;
  }
  else {
    result.erase(result_iterator);
  }
  number = std::move(result);
}
std::ostream &operator<<(std::ostream & os,
                         const Number & num){
  std::for_each(num.number.crbegin(),
       num.number.crend(), [& os](const int n) {
    if (n >= 10){ // If the digit cannot be
       // represented by the characters '0' to '9'
      os << '(' << n << ')';
    }
    else {
      os << n;
    }
  });
  std::cout << " base " << Number::base;
  return os;
}
bool Number::is_palindrome() const{
  return std::equal(number.cbegin(),
    number.cbegin() + ssize(number) / 2,
    number.crbegin());
}
int main(){
  const int limit = 10’000;
  Number::base = 10;
  for (Number number{0}; number < Number{2, 0, 0};
       ++number) {
    Number combined_number = number;
    int iterations = 0;
    while (iterations <= limit) {
      if (combined_number.is_palindrome()) {
        std::cout << number << " becomes "
          << combined_number
          <<  " which is a palindrome after "
          <<  iterations << " iterations\n";
        break;
      }
      combined_number.reverse_and_add();
      ++iterations;
    }
    if (iterations > limit)     {
      std::cout << number
        << " did not become a palindrome within "
        << limit << " iterations\n";
    }
  }
  std::cout << "Finished\n";
}
```

The Standard Report

Guy Davidson reports on the recent face-to-face meeting in Hawai'i.

On November 7th, the third WG21 plenary of 2022 was convened in Kona, Hawai'i. It was the first face-to-face WG21 meeting since Prague in February 2020. It was also our first hybrid meeting, and I would like to start by expressing my appreciation and gratitude for the team of people who made it possible, and briefly describe the process.

Earlier in the year, the committee leadership decided to make hybrid meetings work. An approach was trialled at CppCon in October, which taught us a lot about how to conduct things and about the kind of equipment we would need. Eventually, we ended up with slightly different arrangements in each room. We had zoom meetings for each room, but the cameras and microphones were configured according to the audience and size.

It all worked quite well, really. I spent most of my time in the Library Evolution Working Group (LEWG) where there were a couple of microphones, a camera, and a screen shared for zoom participants containing the topic under discussion. Participants were sufficiently disciplined not to talk over each other or engage in side-conversations, both of which make it very hard for people to participate remotely. The general feedback from all participants was very positive, and although there are still a few kinks to iron out, I foresee general improvement.

You should consider participating remotely. Prior to hybrid meetings you would commit to spending a week away from your day job, arranging flight, accommodation and subsistence, and making yourself useful over the entire period. It is now feasible to simply join those sessions which are relevant to you and which you can meaningfully contribute to.

Time zones and scheduling were problematic. Hawai'i is ten hours behind GMT which meant that any afternoon sessions were out of reach to all but the most heroic. Congratulations must go to Jonathan Wakely, the chair of Library Working Group (LWG) and a UK resident who was available throughout the meeting while remaining at home. While LEWG stuck to its schedule, Evolution Working Group (EWG) was a little more free flowing with its subject matter and some proposals were discussed unannounced.

By the end of the week, 27 straw polls were scheduled for the closing plenary. As you might imagine, being so close to the ship date for the next standard, there were only subtle changes. Core Working Group (CWG) offered 12 straw polls, of which nine were accepting defect reports and amending the working draft appropriately, and three were National Body (NB) comment resolutions. The last of these amends the behaviour of range-based `for` loops.

I have singled this out for closer examination because it resolves a somewhat confusing aspect of this feature. Particularly, the lifetime of temporary objects in the `for`-range-initializer is extended until the end of the loop. This improves the usability of the range-based `for` loop since

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.

Report on Homework Challenge 13 (continued)

I think that if this code was presented to the student's tutor, it would be realised that it was not written solely by the student and that significant assistance was received. Providing the student understands and can adapt the advice given then it is all to the good. We all seek ideas and inspiration from the work of others, after all.

In conclusion

I think that Silas clearly wins on the basis of obfuscation. However, I like James' approach and it has much more potential for development. For anyone who is not entirely up to date with their C++, note that James is using the now authorised thousands separator.

If anyone reading this missed the unstated deadline. Please do send your code in for publication next time.

Homework Challenge 14

This one works far better if you can do it with coloured output, but if you cannot, black and white will do.

I suspect that most readers are familiar with Pascal's Triangle but if you aren't you can find out what you need for this homework challenge at https://en.wikipedia.org/wiki/Pascal%27s_triangle

You only need the first two paragraphs of that entry and the rest might be overload for those whose mathematical studies were long in the past.

The challenge is to write a program that outputs the first 32 lines of Pascal's Triangle correctly formatted but using a finite or modulo (sometimes called a clock or remainder) arithmetic. In these, you carry out normal mathematical operations of add, subtract and multiply (division is more problematical as the answers have to be derived from reversing multiplication rather than using traditional division) and then reduce the answer by the modulus. Confused? Well here are a few examples:

In modulus 7:

$3 + 4 = 0$

$5 + 4 = 2$

$1 - 6 = -5$ (which becomes 2)

$4 \times 4 = 2$ (16/7 gives remainder 2)

For the purpose of this exercise, you only need to deal with addition. The program needs to produce a well formatted output where the entries have been reduced by the selected modulo, from 2 to 20. The more interesting results are where the modulus is a prime number and the resulting patterns are more clearly seen if you can colour code the output (from black for zero to violet for the highest value).

This is an exercise that my maths students used to enjoy doing by hand. It reveals some of the hidden patterns that Pascal's Triangle is richly endowed with.

By the way, if you had not realised, modular arithmetics are related to representation in different bases where you discard all but the units place. However, division is still more problematical – there is always an integer answer but sometimes there is more than one.

Have fun and send your results to me at francis.glassborow@btinternet.com by 1st February 2023, with the subject of the email being HW14.

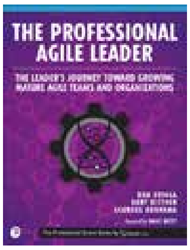
Reviews

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.

The Professional Agile Leader



By Ron Eringa, Kurt Bittner and Laurens Bonnema, published by Addison-Wesley Professional, website: https://www.pearson.com/en-us/subject-catalog/p/professional-agile-leader-the-growing-mature-agile-teams-and-organizations/P200000000238/9780137591510, ISBN: 978-0137591510

Reviewed by Alan R Griffiths

Verdict: Recommended

I'm sure that many of you will, like me, have experience of introducing Agile processes and ideas into organizations. My experience is that it is possible to be effective in this at the scale of a software development team, or of a few teams, and that it gets rapidly harder to make progress as more of the organization is affected. Of course, I've usually seen this from the point of view of a team member or practice lead within a development group.

This book views the organizational transformation from a different perspective: that of Doreen the CEO of an organization embarking on an agile transformation. From this perspective, it explores many of the reasons for the difficulties I've encountered and shows how, with the right leverage, these can be resolved.

Doreen drives the transformation of that organization from a 'traditional' model to an 'agile' one. She is guided along the way by the CEO of a new subsidiary that has already adopted agile thinking. Despite the different perspective, I recognise many of the struggles along the way as points at which organizations I've worked with have failed.

There are some minor failures along the way, but mostly successes; to the extent that it almost seems too easy. Mostly, I have found it far harder to get buy-in from senior management than it appears from the 'conversations' between the heroes of this book. However, the solutions proposed are plausible, the results reasonable and, I'm sure, based upon the experience of the authors. Expanding the discussions to the extent that I have experienced in real life would have inflated the book unreasonably.

This book gives a clear overview of an agile transformation in an organisation with many of the real-life resistances that will be encountered. It should not be mistaken for a complete guide to every problem that will ever be encountered, but it does provide waymarks that will guide the way.

I love it, it is not heavy going and gives me hope of achieving more in the future!

The Standard Report (continued)

destructors of some temporary objects are invoked later. See Listing 1 for an example.

This change will now be detailed in Annex C of the standard, which considers compatibility between older versions and the current version.

Meanwhile, in library, we decided to bring the Library Fundamentals TS series to a close with version three. The rationale is that we now have a train model where new versions are shipped every three years and the need for a separate testing ground is less obvious. Delivering a feature from the TS is simply a matter of importing it into the standard, so there is no real need for it to continue to exist as a staging post. Its existence has slowed the adoption of new library features located therein since it seems to be a lower priority. LEWG will now standardise with confidence rather than tentatively forward to an experimental holding area.

The concurrency TS, on the other hand, saw new features being added, particularly byte-wise atomic `memcpy` from P1478 and asymmetric fences from P1202. The remaining straw polls were NB comment resolutions.

The careful brevity of the work in Kona brings me to the end of this shortened edition of the standard report. In the next edition I will discuss the events of the Issaquah meeting, where we expect to ship C++23 to a waiting world. I will also look at some upcoming proposals for C++26. Do join me then.

```
void f()
{
  std::vector<int> v = { 42, 17, 13 };
  std::mutex m;
  for (int x : static_cast<void>(
    std::lock_guard<std::mutex>(m)), v)
        // lock released in C++ 2020
  {
    std::lock_guard<std::mutex> guard(m);
        // OK in C++ 2020, now deadlocks
  }
}
```

Listing 1

ACCU Information

Membership news and committee reports

View from the Chair

Geoffrey Daniels
chair@accu.org

Over the last few months I have noticed AI models gaining more and more column inches on news websites. I am someone who works in computer vision, an area that heavily utilises machine learning, so I thought it is about time for me to give my view on our future robotic overlords. So, take it away GPT-3…

The brief

Write an article for the ACCU titled a ‘View from the Chair’, containing insights into the world of software engineering. This article should be about 500 words long and focus on dissecting the topic: ‘recent machine learning advances’. Include information about the following topics: explainable AI, trusting closed source systems, GPT-3 for text generation, Stable Diffusion for image synthesis, NERF for 3D modelling, solving advent-of-code, and will AI replace software engineers.

The article

As the chair of the ACCU, I have had the opportunity to observe and participate in the rapidly evolving world of software engineering, particularly in the realm of machine learning. In recent years, we have seen a multitude of advances in this field, and it can be difficult to keep up with all the new developments. In this article, I will delve into some of the most notable recent machine learning advances and explore the implications they have for the software engineering industry.

One of the most buzzed-about advancements in machine learning is explainable AI, which refers to artificial intelligence systems that can provide clear, understandable explanations for their decision-making processes. This is an important development because it allows us to trust closed source systems, which are systems whose inner workings are not visible to us. With explainable AI, we can be confident that the system is making decisions based on logical reasoning, rather than relying on black box algorithms that we cannot fully understand.

Another notable advancement is the use of GPT-3 for text generation. GPT-3, or Generative Pre-trained Transformer 3, is a language processing AI developed by OpenAI that can generate human-like text with a high level of accuracy. This has the potential to revolutionize industries such as journalism, where GPT-3 could be used to generate news articles or reports. However, it also raises concerns about the potential for AI to replace human writers and journalists.

Stable Diffusion is another machine learning technology that has garnered attention in recent years. This AI system is designed for image synthesis, meaning it can generate realistic images based on input data. This technology has the potential to revolutionize industries such as film and video game development, where it could be used to create realistic 3D environments and characters.

NERF, or Neural Radiance Fields, is a machine learning system designed for 3D modelling. It can generate detailed 3D models of objects based on input data, and has been used to create realistic 3D models of everything from furniture to human faces. This technology has the potential to greatly streamline the 3D modelling process, making it faster and more efficient.

Another recent machine learning development that has garnered attention is the use of AI to solve advent-of-code challenges. Advent-of-code is a series of programming challenges that are released every December, and AI systems have been used to solve some of these challenges with impressive speed and accuracy. While this is a great achievement for AI, it does raise concerns about the potential for AI to replace human programmers in certain tasks.

Finally, the question on everyone’s mind is: will AI replace software engineers? While AI has the potential to automate certain tasks and streamline processes, it is unlikely to fully replace human software engineers. While AI can handle certain tasks with greater speed and accuracy, it lacks the creativity and problem-solving skills that humans possess. In the end, it is likely that AI and software engineers will work together, with AI handling certain tasks and humans handling others.

In conclusion, recent machine learning advances have the potential to revolutionize the software engineering industry. From explainable AI to NERF for 3D modelling, these technologies have the power to streamline processes and automate certain tasks. However, it is important to keep in mind that AI is not a replacement for human software engineers, and that the two will likely work together in the future.

The reaction

Both terrifying and amazing.

Best Articles 2022

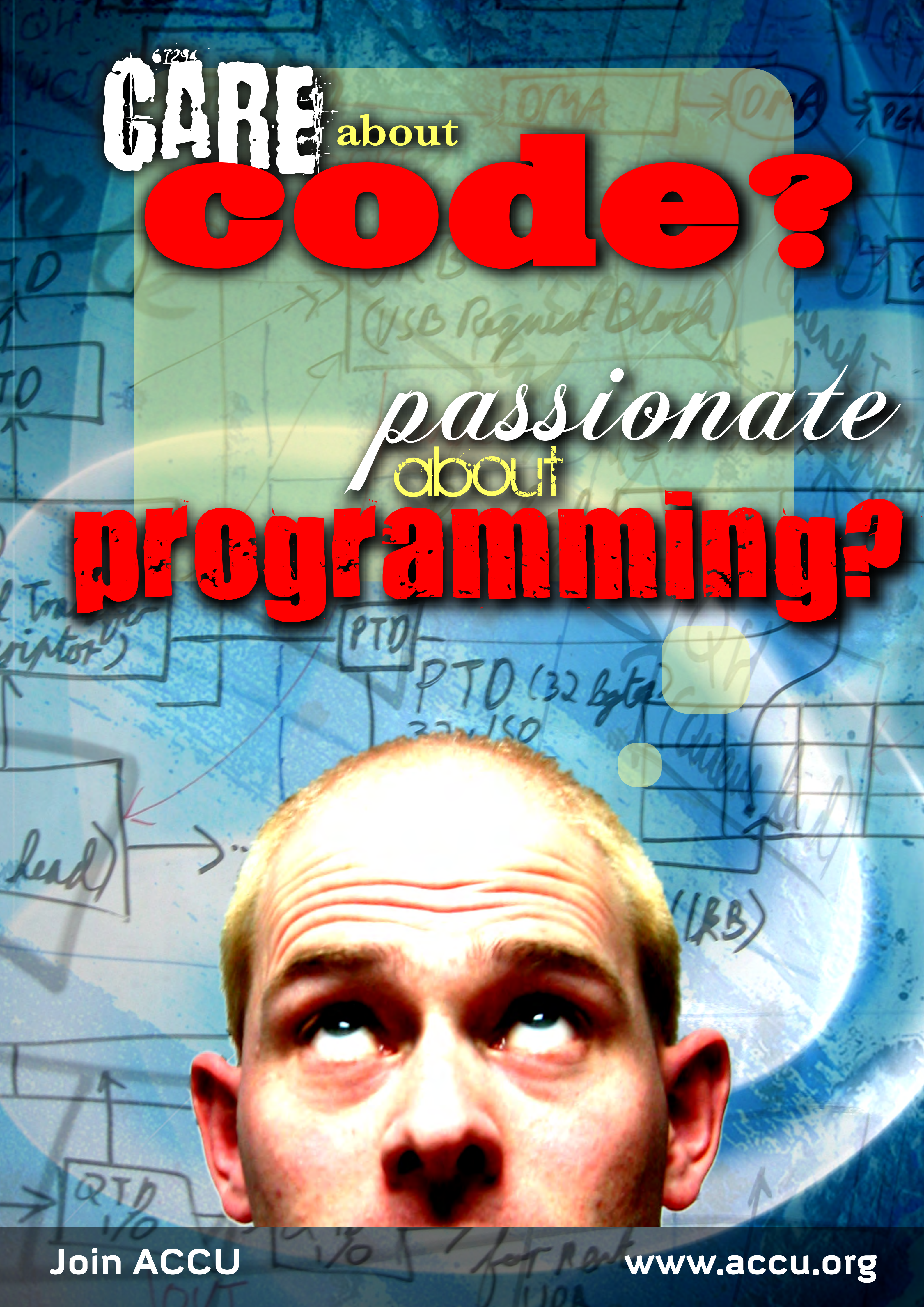
Vote for your favourites:

- Best in *CVu*
- Best in *Overload*

Select up to 3 favourites from each journal.

Voting open online at:

https://www.surveymonkey.co.uk/r/ZJ3TF9P


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

Join ACCU

Run by programmers for programmers,
join ACCU to improve your coding skills

- A worldwide non-profit organisation
- Journals published alternate months:
  - *CVu* in January, March, May, July, September and November
  - *Overload* in February, April, June, August, October and December
- Annual conference
- Local groups run by members

Join now!
Visit the website

professionalism in programming

www.accu.org