the magazine of the accu

www.accu.org

{cvu}

Volume 30 • Issue 4 • September 2018 • £3

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 30 Issue 4
September 2018
ISSN 1354-3164
www.accu.org

Precision Engineered

I was recently watching a TV program about re-conditioning and repairing old, valuable and much-loved items. In one episode, a badly damaged chair is brought to the workshop, and it's in a very sorry state. It's made partly of wood, and broken into two pieces.

The chair is a much sought-after item, and is a very elegant, cleverly designed piece of furniture. Each component part is made from the finest, most exclusive materials of the time, precision-engineered by expert craftspeople, all brought together to make more than just a very nice chair: it's a piece of art, and a fine example of design engineering.

But now, it's broken and unusable. The problem appears to have been caused by a previous repair to an arm mounting. The failure of this component whilst the chair was being used normally caused a catastrophic fault in the chair's wooden back support, which has snapped irreparably. Stresses in one small, but faulty, part of the chair finally caused another part of the chair to reach breaking point.

To compound the problems, the broken back support is made from a type of wood that is no longer available to buy. To make the chair functional again, it must be replaced, as it cannot be repaired, and so a different type of wood has to be used. A replacement wooden back is found, and although the colour isn't quite an exact match, it's at least as strong, of a similar style, and judged good enough.

The chair is finally back in one piece, but it's taken a large amount of time and effort, and the end result isn't *exactly* like the chair as it was initially designed. The original makers never foresaw that the exclusive materials they used might not be available in the future, and didn't anticipate that a slight defect in one component could have such serious consequences.

Does any of this sound familiar to anyone else?

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 30.5: 1st October 2018
C Vu 30.6: 1st December 2018

Overload 147: 1st November 2018
Overload 148: 1st January 2019

WRITE FOR C VU

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

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

ADVERTISE WITH US

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

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

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

COPYRIGHTS AND TRADE MARKS

The Ghost of a Codebase Past

Pete Goodliffe learns lessons by reviewing his own old code.

I will live in the Past, the Present, and the Future.
The Spirits of all Three shall strive within me.
I will not shut out the lessons that they teach!

~ Charles Dickens, *A Christmas Carol*

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

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

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

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

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

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

> Looking back at your older code will inform you about the improvement (or otherwise) in your coding skills.

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

Revisiting your old code can be ... like visiting an ageing, distant relative you don't see very often

Presentation

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

Different tribes of C++ programmers, for example, follow different presentation schemes. Some developers follow the standard library scheme:

```
struct standard_style_cpp
{
  int variable_name;
  bool method_name();
};
```

Some have more Java-esque leanings:

```
struct JavaStyleCpp
{
  int variableName;
  bool methodName();
};
```

And some follow a C# model:

```
struct CSharpStyleCpp
{
  int variableName;
  bool MethodName();
};
```

A simple difference, but it profoundly affects your code in several ways.

Another C++ example is the layout of member initialiser lists. One of my teams moved from this traditional scheme:

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

to a style that places the comma separators at the beginning of the following line, thus:

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

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

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

As long as a style is employed consistently in your codebase, this is really a trivial concern and nothing to be embarrassed about. Individual coding styles rarely make much of a difference once you get used to them, but inconsistent coding styles in one project make everyone slower.

The state of the art

Most languages have rapidly developed their in-built libraries. Over the years, the Java libraries have grown from a few hundred helpful classes to

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 veritable plethora of classes, with different skews of the library depending on the Java deployment target. Over C#'s revisions, its standard library has also burgeoned. As languages grow, their libraries accrete more features.

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

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

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

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

with its inherent potential for bugs, would have become:

```
List<string> list = new List<string>();
list.Add("Foo");
list.Add(3); // compiler error - nice
```

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

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

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

Idioms

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

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

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

Many years ago, I worked with a team of C programmers moving (well, shuffling slowly) towards the (then) brave new world of C++. One of their initial additions to a new codebase was a **max** helper macro:

```
#define max(a,b) ((a)>(b)) ? (a) : (b))
// do you know why we have all those brackets?

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

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

Listing 1

```
template <typename T>
inline T max(const T &a, const T &b)
{
  // Look mum! No brackets needed!
  return a > b ? a : b;
}
void better_example()
{
  int a = 3, b = 10;
  // this would have failed using the macro
  // because ++a would be evaluated twice
  int c = max(++a, b);
}
```

In time, someone revisited that early code and, knowing more about C++, realised how bad it was. They rewrote it in the more idiomatic C++ shown here, which fixed some very subtle lurking bugs (see Listing 1).

The original version also had another problem: wheel reinvention. The best solution is to just use the built-in **std::max** function that always existed. It's obvious in hindsight:

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

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

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

That's a simple example, but as languages gain new features (e.g., lambdas) the kind of idiomatic code you'd write today may look very different from previous generations of the code.

Design decisions

Did I *really* write that in Perl; what was I thinking?! Did I *really* use such a simplistic sorting algorithm? Did I *really* write all that code by hand, rather than just using a built-in library function? Did I *really* couple those classes together so unnecessarily? Could I *really* not have invented a cleaner API? Did I *really* leave resource management up to the client code? I can see many potential bugs and leaks lurking there!

As you learn more, you realise that there are better ways of formulating your design in code. This is the voice of experience. You make a few mistakes, read some different code, work with talented coders, and pretty soon find you have improved design skills.

Bugs

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

Conclusion

No space of regret can make amends
for one life's opportunity misused.
~ Charles Dickens, *A Christmas Carol*

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

Does this kind of thing actually *matter*? If your old code isn't perfect, but it works, should you do anything about it? Should you go back and

The Rich Get Richer

The Baron has another dice game and invites you to take a wager.

Sir R-----! I must say that it is a relief to have the company of a fellow nobleman in these distressing times. That I have had to sell not one, but two of my several hundred antiquities to settle the burden of tax that this oppressive democracy has put upon me, simply to enrich slugabeds I might add, is quite intolerable!

Come, let us drown our sorrows whilst we still have the means to do so and engage in a little sport to raise our spirits.

I have a fancy for a game that I used to play when I was the Russian ambassador to the Rose Tree Valley commune. Founded by the philosopher queen Zway Remington as a haven for downtrodden wealthy industrialists, it was the purest of pure meritocracies; no handouts to the idle labouring classes there!

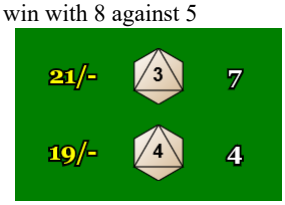
We would spend our evenings at lavish feasts, with copious supplies of the finest comestibles produced by a miraculous device that Queen Remington had, quite literally, pulled out of thin air. Once sated, we would retire to discuss at length the great civil works with which they planned to make a veritable paradise upon Earth and, at somewhat greater length, the objective truth that living by her philosophy should be the means by which the better part of humankind might free itself from the tyranny of the masses.

Tragically, some months after I had been recalled to the Empress's court, I learned that the entire community had succumbed to an epidemic of cholera, the occurrence of which defies all rational explanation.

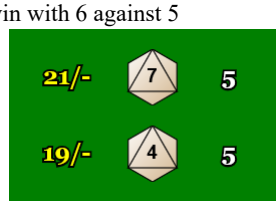
But let us not dwell upon the misfortunes that those such as we must ever endure and instead commence our sport!

At the cost of one coin you shall begin the game with a score of one point and I with a score of two. At each turn, you shall cast an eight-sided die and increase your score by one point should its face be no greater than it, and I shall do likewise. If your score exceeds mine before I reach a goal of eight points then you shall have the game and a prize of five coins.

- The Baron will win with 8 against 5



- Sir R----- will win with 6 against 5



When I told that layabout student of this game, he began lamenting that his physical condition was such that he should probably be excused participation in the construction of a rehearsal room. Quite why he and his low-born fellows imagine that the nobility could be fooled by their transparent excuses to shirk any manner of honest toil is entirely beyond my comprehension!

Enough of that wretch! Here, take another glass whilst you decide whether or not you shall play! ■

Baron M

Courtesy of www.thusspakeak.com

BARON M

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

The Ghost of a Codebase Past (continued)

'adjust' the code? Probably not – *if it ain't broke don't fix it*. Code does not rot, unless the world changes around it. Your bits and bytes don't degrade, so the meaning will likely stay constant. Occasionally a compiler or language upgrade or a third-party library update might 'break' your old code, or perhaps a code change elsewhere will invalidate a presumption you made. But normally, the code will soldier on faithfully, even if it's not perfect.

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

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

Questions

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

Pete's latest book, *Becoming a Better Programmer*, is published by O'Reilly. It's available at http://shop.oreilly.com/product/0636920033929.do (and all good book stores).

On Francis's Challenge #4

Andreas Gieriet presents his solution (ab-)using expression- and jump-statements.

This is the winning entry to Francis's Challenge #4, presented in *CVu* 30.3. Both Francis and Steve felt this entry deserved an article of its own.

I like this kind of challenge! Especially Francis's sentence "I cannot think of a way to do this in C other than..." triggered my curiosity. It asks for lateral thinking and invites me to explore the (sometimes dark) corners of the languages C and C++.

For the fun of it, I allow myself to do rather dirty but 'legal' hacks here and there. The shown approaches are not exhaustive – there are for sure many more possible solutions.

As a disclaimer: I do not suggest that any of the shown code snippets are good practice nor that they should be used in productive code. The snippets show what is possible, not what is desirable.

The challenge

To the challenge: write a (C/C++) program that prints integral numbers from two run-time boundaries 'first' to 'last' (inclusive). The only constraints are not to use anything classed as either an *iteration_statement* (**`while`**, **`do-while`**, **`for`**) or a *selection_statement* (**`if`**, **`switch`**). There are explicitly no other constraints (so, use that freedom!).

The tools

What language and library tools do we have at hand to circumvent the constraints?

- language: all but the *selection_statement* and *iteration_statement*. E.g. C/C++: *labeled_statement*, *expression_statement*, *jump_statement*; C++: *try_block*, etc.

  See [2] and [3] and Figure 1, which shows C++ statements.

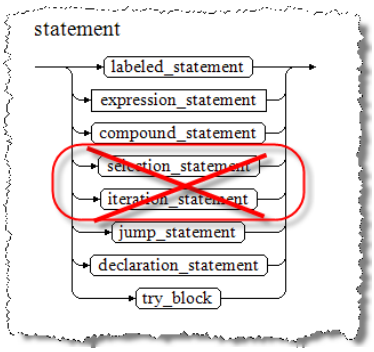
  Note: for the purpose of simplicity, I refer to C++ syntax descriptions, knowing that C is in most regards a subset of those, i.e. for the scope of this text, C is considered to be covered by these syntax references).

- standard libraries: all libraries, types and functions. E.g. C/C++: **`assert`**, **`exit`**; C++: **`std::for_each`**, etc.

  See [2] and [3].

- compiler generated implicit code: all. E.g. C++: constructors, etc.

Figure 1



Let's dive into the world of missing *iteration_statements* and *selection_statements*!

We will talk about short-circuit evaluation and the comma operator to mimic **`if`**, about **`goto`** to produce some iteration, about various recursions as replacement for iterations, touch lookup tables and function factories, and finally get into some C++ approaches like **`range iterator`** and benefiting from constructors.

All this is mingled with code snippets, standard references and some more exotic ideas like **`assert`** to terminate a program and **`longjmp`** to mimic iteration.

Let's start!

Compile-time versus run-time boundaries

If we had compile-time boundaries for printing the range of numbers, we could use template 'magic'.

With run-time boundaries, we cannot use any template 'magic' directly AFAIK.

Since we are bound to run-time values, we must first read these two values. No constraint is given if we read these values via command line arguments or via some input stream. Neither is there any error handling required.

BTW: it is not even required that the program terminates or that it terminates gracefully. E.g. it might loop forever once all the numbers are printed, or it might crash when the task has completed.

Conclusion: Template magic is not of any direct use in this challenge.

Short-circuit evaluation

Conditional code execution is needed e.g. to read the values from the command line, or to conditionally terminate the print loop or to terminate recursion, etc.

C/C++ provides the short-circuit evaluation of the logical_and_expression (**`a && b`**) as well as of the logical_or_expression (**`a || b`**) as well as of the conditional operator (**`c ? a : b`**)

See [2], section 5.14 Logical AND operator:

> [...] The **`&&`** operator groups left-to-right. The operands are both contextually converted to bool [...]. The result is true if both operands are true and false otherwise. Unlike &, && guarantees left-to-right evaluation: the second operand is not evaluated if the first operand is false. [...]

See [2], section 5.15 Logical OR operator

> [...] The **`||`** operator groups left-to-right. The operands are both contextually converted to bool [...]. It returns true if either of its operands is true, and false otherwise. Unlike |, || guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the first operand evaluates to true. [...]

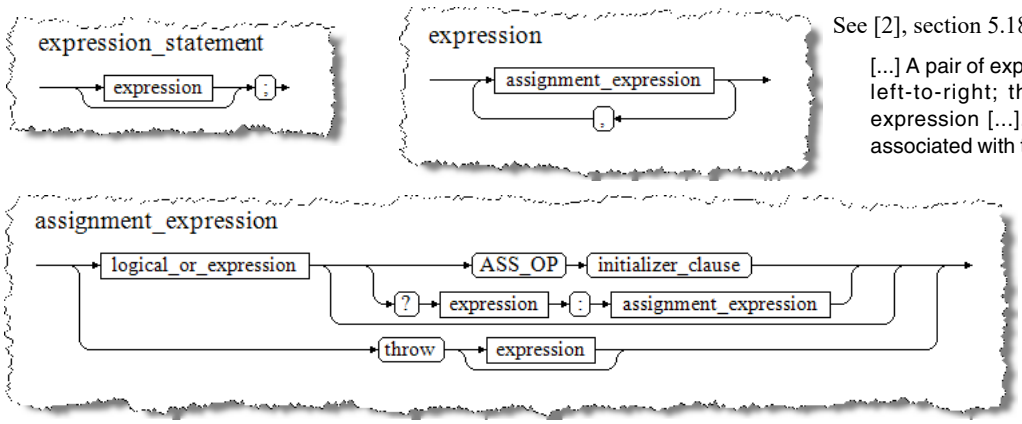
See [2], section 5.16 Conditional operator

> [...] Conditional expressions group right-to-left. The first expression is contextually converted to bool [...]. It is evaluated and if it is true, the result of the conditional expression is the value of the second expression, otherwise that of the third expression. Only one of the second and third expressions is evaluated. Every value computation and side effect associated with the first expression is sequenced before every value computation and side effect associated with the second or third expression. [...]

ANDREAS GIERIET

Andreas Gieriet holds an MSc in electrical engineering, and freelances in several roles in the SW engineering food chain. He has a passion for lecturing and still gets pleasure from "finding things out" after many years. He can be contacted at andreas.giriert@externsoft.ch

Figure 2



See [2], section 5.18 Comma operator.

> [...] A pair of expressions separated by a comma is evaluated left-to-right; the left expression is a discarded-value expression [...]. Every value computation and side effect associated with the left expression is sequenced before every value computation and side effect associated with the right expression. The type and value of the result are the type and value of the right operand [...]

Conclusion: The comma operator is useful in connection with the short-circuit evaluation under the constraints that no `if` nor `switch` is allowed.

We can take benefit of that well-defined behaviour, e.g.

```
argc == 3 || print_usage_and_exit(argv);
```

where, for example

```
bool print_usage_and_exit(const char* argv[])
{
  printf("usage: %s [first] [last]\n", argv[0]);
  exit(1);
  return true; // not reached but returns boolean
  // for convenience in short-circuit expressions
}
```

Note: There is no need to have an `if` around `argc == 3` nor to have the result assigned to any variable. The languages allow for this construct (see Figure 2: *expression_statement* in top-left, *expression* in top-right and *assignment_expression* at bottom).

For details on the `exit` function, see [2], section 18.5, paragraph 8.

See [2], section 6.2.

> [...] Expression statements have the form [...]
>
> $\texttt{expression}_{\texttt{opt}}$;
>
> The expression is a discarded-value expression [...] An expression statement with the expression missing is called a null statement. [...]

And [2], section 5, paragraph 11.

> [...] In some contexts, an expression only appears for its side effects. Such an expression is called a discarded-value expression. The expression is evaluated and its value is discarded. [...]

Note: Short-circuit evaluation is useable for **expression** statements only.

E.g. the following is not possible since `return` is a jump-statement and not an expression(_statement):

```
argc == 3 || return; // won't compile
```

Note also, that an `else` or `else-if` part can be emulated e.g. by the respective `not-if` condition as prefix for an `else` part.

Caution: Do not get confused by expression statements which form an assignment. They are still discarded-value expressions (even there is an assignment, e.g. v = 5;). The reason is that an assignment itself yields a value which allows to chain assignments (e.g. a = b = c = 5;). The left-most assignment yields its value which then is discarded.

Conclusion: Short-circuit evaluation can emulate some `if` constructs as long as the code after the condition is an expression(-statement) itself.

Comma operator

The comma operator is useful to sequence multiple expression-statements or to have non-return functions in a (short-circuit) boolean expression. E.g.

```
argc == 3
|| (printf("usage: %s [first] [last]\n, argv[0]),
exit(1), true);
```

Parse arguments by short-circuit evaluation and comma-operator

A possible C solution to parse command line arguments would be as shown in Listing 1. (C++ was similar but with C++-ish includes, io, bool, and namespaces.)

Loop forever by label and goto

Loop 'forever' is the basic iterative control flow for our challenge. Since no *iteration_statement* is allowed, we have to divert to other means.

One is a label-statement (`label:...`) and one of the jump-statements (`goto label;`). E.g.

```
  int first = ...
  int last = ...
loop:
  first <= last || (exit(0), 1);
  //or for C++ with (..., true) instead of (..., 1)
   printf("%d\n", first);
  first < last || (exit(0), 1);
  //needed for the case where last == INT_MAX
  first++;
  goto loop;
```

Note: The unconstrained version with a `while`-loop (or any other *iteration_statement*) would also need special handling of the `last == INT_MAX` case, i.e. you must not increment beyond `INT_MAX`. (E.g. see Listing 2.)

Listing 1

```
#include <stdio.h>
#include <stdlib.h>
void print_usage_and_exit(const char* argv[])
{
  printf("usage: %s [first] [last]\n", argv[0]);
  exit(1);
}
int parse_int(int argc, const char* argv[],
  int pos)
{
  argc > pos && pos > 0 ||
    (print_usage_and_exit(argv), 1);
  char* next = 0;
  int value = strtol(argv[pos], &next, 10);
  next && !*next || (print_usage_and_exit(argv),
    1);
  return value;
}
int main(int argc, const char* argv[])
{
  int first = parse_int(argc, argv, 1);
  int last = parse_int(argc, argv, 2);
  // more to come here
}
```

Listing 2

```
int i = first;
while (i < last)
{
  printf("%d\n", i);
  i++;
}
if (i == last)
{
  printf("%d\n", i);
}
```

Conclusion: The label/goto pair form a robust and quite straight forward solution for the given challenge.

Exotic: How about assert(...) to let evaluate the stop condition?

Instead of the short-circuit evaluation, we could also use the **assert** function to terminate the program. This is not so graceful a termination but it does not violate the constraints, e.g.

```
    int first = ...
    int last = ...
  loop:
    assert(first <= last);     // terminates the
          // program by a crash if the condition
          // is not satisfied
    printf("%d\n", first);
    assert(first < last);
      // covers last == INT_MAX
    first++;
    goto loop;
```

Note: This usually only works in Debug mode. See also [2], section 19.3 Assertions, as well as [3], section 7.2 Diagnostics <assert.h>.

Conclusion: Fulfils the constraints, not so nice termination.

Exotic: How about to not terminate the program at all?

We could also let the program run forever and print the range and then do nothing other than looping idle, e.g.

```
    int first = ...
    int last = ...

    int verbose = 1;
  loop:
    verbose = verbose && (first <= last);
    verbose && printf("%d\n", first);
    first < last || (verbose = 0);
      // INT_MAX safe end handling
    first++;
    goto loop;
```

Conclusion: Not user friendly solution since it (silently) does not terminate at all.

Simple recursion

Recursion is the alternative to iteration. In its simplest form, it requires a stop condition and a conditional call to itself with modified parameters. E.g.

```
int print_range_recursive(int current, int last)
{
  current <= last && printf("%d\n", current);
  current <  last &&
    print_range_recursive(current+1, last);
  return 1;
}
```

Note: Even there are no local variables, the return address must still go on the stack. This means that for large ranges, you probably run out of memory or run into a stack overflow condition.

Note: the stack size is assumed to be linearly proportionate to the range, e.g. a range of 10 to the power of 9 results in a stack usage proportional to 10 to the power of 9.

Conclusion: Not robust for large ranges.

Tail recursion

Tail recursion is the ability of the compiler to not call the function recursively under certain conditions but rather perform an implicit **goto** to the function label and only return from the function on its stop condition.

The construct is usually as follows (assuming the compiler is capable to detect this and act accordingly):

```
void f(args)
{
  // return if stop...
  // ...do some stuff...
  f(args+1); // must be last statement:
             // that's where the term "tail
             //   recursion" comes from
}
```

The trouble here: how to do a conditional return which is not the last statement (under the given challenge's constraints: no **if**).

In C, I see no decent solution other than to exit the program at that point, e.g.

```
void print_range_tail_recursive(int current,
  int last)
{
  current <= last || (exit(0), 1);
  printf("%d\n", current);
  current < last || (exit(0), 1);
  print_range_tail_recursive(current+1, last);
    // assuming the compiler optimizes
    // this to tail recursion
}
```

In C++ we could use a **try-catch** block, like Listing 3.

Note: If the compiler applies tail recursion, this solves the resource usage of the simple recursion solution above.

Listing 3

```
void print_range_tail_recursive_cpp(int current,
  int last)
{
  try
  {
    current <= last || (throw 0, true);
    // or give a more decent exception name/value

    printf("%d\n", current);
    current < last || (throw 0, true);
    // safe end handling
  }
  catch(...)
  {
    return;
  }
  print_range_tail_recursive_cpp(current+1,
    last);
// assuming the compiler optimizes this to
// tail recursion
}
```

Listing 4

```
int print_range_balanced_by_two_recursive(
  int current, int last)
{
  int cut = (current + last) / 2; // Troublesome!
                                  // See notes below.
  current >  last // nop
  || current == last && printf("%d\n", current)
  || print_range_balanced_by_two_recursive
    (current, cut)
  && print_range_balanced_by_two_recursive
    (cut + 1, last)
  ;
  return 1;
}
```

Conclusion: If the compiler supports and actually applies here the tail recursion, stack usage is constant, i.e. independent of the range.

Balanced recursion

The idea is to replace the simple recursion by cutting the range into two halves in each recursion and then traverse each half recursively in the same way. This results (in absence of tail recursion) in maximum stack usage proportional to log(range)/log(2). E.g. for a range of 10 to the power of 9, the maximum function call depth would be about 30 (compared to 10 to the power of 9). For example, see Listing 4.

Note: the term (current + last)/2 might get out of range, e.g. `(INT_MAX + INT_MAX)/2` has an intermediate result of twice `INT_MAX`!.

This can be avoided by a bit more elaborate expressions like the following below: each term never exceeds the max range and neither does the overall sum.

```
int cut = current/2 + last/2 +
  (current%2 + last%2)/2;
```

But other trouble lies ahead: This only works for positive limits (x/2 rounds towards 0 while x >> 1 rounds towards `INT_MIN`; e.g. -3/2 = -1 while -3 >> 1 = -2). The correct solution is to use arithmetic right-shift, since this simply cuts off one bit on the right and preserves the sign. So, a version which also works for negative limits and respects full number range is:

```
int cut = (current>>1) + (last>>1) +
  (((current&1) + (last&1))>>1);
```

But unfortunately: the right-shift operator is 'implementation defined' if working on signed negative values! See [2], section 5.8 Shift operators, paragraph 3:

> [...] The value of E1 >> E2 is E1 right-shifted E2 bit positions. [...] If E1 has a signed type and a negative value, the resulting value is implementation-defined. [...]

Life could be so easy... On most machines this will work. Sometimes you have to make a decision to still use 'implementation defined' features of a language and to make it safe, add assertions to document your assumptions. E.g.

```
// implementation defined: arithmetic shift with
// signed negative values see C++ standard
// (http://www.open-std.org/jtc1/sc22/wg21/)
// section 5.8 Shift operators, paragraph 3
assert(-2 >> 1 == -1);
assert(-3 >> 1 == -2);
assert(-4 >> 1 == -2);
```

Conclusion: Robust and (almost) compiler-independent solution which heavily reduces stack usage.

Listing 5

```
void end(int* current, int last)
{
  exit(0);
}
void print_and_increment(int* current, int last)
{
  printf("%d\n", *current);
  (*current)++;
}
typedef void (*action_t)(int* current, int last);
action_t action[] =
{
  print_and_increment,
  end,
};
...
 first = ...
 last = ...

loop:
  action[first <= last ? 0 : 1](&first, last);
  goto loop;
```

State lookup table

The idea is to have a state (comparison of first versus last value) which chooses between actions. The first simple version was to have the conditions map into a table index.

In C, this works out-of-the-box (the condition evaluates to 0 or 1, which can be used as index into the table).

Caution: you must make sure that the condition expression yields an index of 0 and 1 (and not 0 and != 0).

In C++ (and C) you might translate by the ternary operator from the condition (which leads to a boolean value) to the index. E.g.

```
first <= last ? 0 : 1
```

An example of a condition to index translation is in Listing 5.

If you want to do more elaborate tables, e.g. with safe end handling, you can do something like Listing 6.

Listing 6

```
void end(int* current, int last)
{
  exit(0);
}
void print_and_end(int* current, int last)
{
  printf("%d\n", *current);
  end(current, last);
}
void print_and_increment(int* current, int last)
{
  printf("%d\n", *current);
  (*current)++;
}
typedef void (*action_t)(int* current, int last);
action_t action[] =
{
  print_and_increment,
  print_and_end,
  end,
};
...
  first = ...
  last = ...
loop:
  action[first < last ? 0 : first == last ? 1
    : 2](&first, last);
  goto loop;
```

Listing 7

```
// ...elided the function definitions from above
// example...
typedef void (*action_t)(int* current, int last);
action_t get_action(current, last)
{
  action_t action = end;
  current < last && (action =
print_and_increment);
  current == last && (action = print_and_end);
  return action;
}
...
  first = ...
  last = ...
loop:
  get_action(first, last)(&first, last);
  goto loop;
```

Function factory

A similar approach to the above was to have a function returning an action based on the condition. (See Listing 7.)

Listing 8

```
#include <iostream>
#include <algorithm>
#include <cstdlib>
using namespace std;
class Range
{
public:
  class It
  {
    private:
      friend class Range;
      It(int first, int last) : current_{first},
        last_{last}, isEnd_{first > last}
      { }
      It(int last)
      : current_{last}, last_{last}, isEnd_{true}
      { }
    public:
      It& operator++()
      {
        isEnd_ = isEnd_ || current_ == last_;
        isEnd_ || ++current_;
        return *this;
      }
      int operator*() const { return current_; }
      bool operator==(It const& other) const
      {
        return isEnd_ &&  other.isEnd_
          || !isEnd_ && !other.isEnd_
          && last_ == other.last_ && current_
          == other.current_;
      }
      bool operator!=(It const& other) const
      { return !operator==(other); }
    private:
      int current_;
      int last_;
      bool isEnd_;
  };
  Range(int first, int last): begin_{first,
    last}, end_{last} {}
  It const& begin() const { return begin_; }
  It const& end()   const { return end_; }
private:
  It const begin_;
  It const end_;
};
...
  int first = ...
  int last = ...
  Range range{first,last};
  for_each(range.begin(), range.end(),
    [](int const& n) { cout << n << endl; });
    ...
```

Listing 8 (cont'd)

C++ range iterator

An iterator mimics pointer logic (++p, p != end, *p) to allow the traversing of containers e.g. by means of **std::foreach**. The standard libraries currently do not provide any range class (AFAIK). So, we can build our own and still comply with the constraints of the given challenge.

Special care is taken again for safe end handling: an **isEnd_** flag is needed since otherwise incrementing beyond the last (e.g. **INT_MAX** value) might break the logic. See Listing 8.

Note: Some non-standard libraries like boost provide ranges, though.

Conclusion: Provides iterators on a range which mimics a container without actual elements.

Constructors of an array

Let the constructor print the items. See Listing 9.

Conclusion: Not so robust since inadvertently constructed Items might disturb the output.

Exotic: longjmp

The standard libraries provide the capability to jump around beyond the capabilities of pure label/**goto** pairs.

See the comments in the code below to get the basics of **longjmp**. See also [3], section 7.13 Nonlocal jumps <setjmp.h>.

Note: I elided safe end handling here since it is complex enough with the **setjmp**/**longjmp** code anyway. See Listing 10.

Listing 9

```
...
int first = 0;
int last = 0;

struct Item
{
  Item()
  {
     static bool verbose = true;
       // safe end handling...
     verbose = verbose && first <= last;
     verbose && printf("%d\n", first);
     first < last && (first++, true)
       || (verbose = false);
  }
};
int main(int argc, const char* argv[])
{
  first = parse_int(argc, argv, 1);
  last = parse_int(argc, argv, 2);
  first <= last
    || (print_usage_and_exit(argv),  1);
  auto data = new Item[last-first+1];
}
```

A Guide to Group Decision Making

Glen Stark advocates an approach to promoting team harmony.

When groups of people gather together to undertake a project, decisions have to be made that affect the group. There are a number of ways that these decisions can be made, some of them are better than others.

In my experience as a software engineer, I have found that the decision-making process is usually poor. The agile manifesto and agile processes have had a transformative effect on how software engineering is conducted. The Agile Manifesto is nine lines long. Two of those lines concern group decision making: "Individuals and interactions over processes and tools" and "Customer collaboration over contract negotiation" [1]. One of the twelve principles behind the Agile Manifesto is "The best architectures, requirement, and designs emerge from self-organizing teams." [2] Clearly the creators of the Agile Manifesto were concerned with collaboration, but most 'Agile' processes scarcely touch, or completely avoid, the topic of group decision-making and group dynamics.

Given the inefficiencies, stress, and sometimes dire consequences that arise out of our poor group decision-making, one would think we would study the topic in our basic education. Sadly, I've never met a person who has had such a course.

This is my attempt to draw some deliberate attention to how teams are making decisions. I identify some good and bad strategies for decision-making and touch on the pros and cons of each of them. My criteria for judging processes are the time and effort costs of the process, the impact on the team's health, happiness, and productivity (which are strongly correlated), and the quality of the decision achieved. I hope that readers will find obvious ways to integrate these thoughts into their existing processes.

Useful decision making strategies

These are the decision making strategies that I consider useful and productive for groups of people. All of these methods work well on a small scale – perhaps for a group of two to a couple of hundred people.

GLEN STARK

Glen is a human being, working as a software engineer on planet Earth. He cares about people, simplicity, reliability and efficiency. He likes to solve problems, ideally without introducing too many new ones in the process. He can be reached at mail@glenstark.net.

On Francis's Challenge #4 (continued)

Note: Use of this in C++ is has some constraints regarding object construction/destruction. See [2], section 18.10, paragraph 4.

Conclusion: Really not the way to do it, I mean, it works, but it's 'goto squared'! ☺

Listing 10

```
#include <limits.h> // see [3], section 5.2.4.2.1
                    // Sizes of integer types
<limits.h>
...
int main(int argc, const char* argv[])
{
  // if not volatile, the value might be cached
  // in a register
  volatile int first = parse_int(argc, argv, 1);
  int last = parse_int(argc, argv, 2);
  // INT_MAX not supported in this code snippet
  last < INT_MAX
    || (print_usage_and_exit(argv), 1);
  // opaque variable for the context where to
  // jump back (like a label)
  jmp_buf jmp_ctx;
  // 0 = init,
  // else came back from "longjmp(..., 1)"
  // this mimics the label
  int jmp_return = setjmp(jmp_ctx);
  // do action (e.g. "printf") if came from
  // "longjmp(..., 1)" since !=0
  jmp_return && printf("%d\n", first++);
  // if first <= last, goto "setjmp" location
  first <= last && (longjmp(jmp_ctx, 1), 1);
}
```

Epilogue

Working on the challenge was fun! I'm convinced that there are many more ways than the ones described above. E.g. an esoteric one might be to use `bsearch` for ranges smaller than 31 values...

I mainly focussed on C implementations. I wonder what (other) C++ solutions for the given challenge are around!

Thanks to Francis again for triggering my synapses. ■

References

For those who want to get into more details, the following sources might be of some use:

[1] https://www.onlinegdb.com/online_c_compiler
Online C/C++ compiler which allows for the ad hoc try out of code in various languages and dialects without the need to install any tool chain.

[2] http://www.open-std.org/jtc1/sc22/wg21/
Lists among other documents the latest publicly available C++ standard draft (e.g. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf).

[3] http://www.open-std.org/jtc1/sc22/wg14/www/standards.html
Lists among other documents the latest publicly available C standard draft (e.g. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf).

[4] http://www.externsoft.ch/media/swf/cpp11-iso.html
Html linked syntax productions which I compiled several years ago for C++ 1998 and 2011 variants.

[5] http://dotnet.jku.at/applications/Visualizer/
Nice EBNF visualization tool. I used it to generate the figures in this text.

- **consensus**: The group debates a decision until concurrences is reached – everyone agrees the decision is the best one on which consensus can be reached. The chance of reaching consensus decreases dramatically as groups grow, but the process can still be useful.
- **democracy**: The group votes on the decision. There are a wide range of possibilities for democratic decisions, but the most common is simple majority.
- **expertise-based authority**: One of the group (or perhaps a subgroup) has expertise in a particular topic, so the group agrees to defer to the obvious expert.
- **stalemate strategies**: Consensus can fail to be achieved. Democracy can be foiled by even numbers. You need strategies for these cases, like coin tossing, taking turns, or resorting to an outside authority.

While each of these strategies deserves deep consideration, it is helpful to attempt an overview of how these strategies compare and interact. Perhaps the most important thing to consider is that each strategy has value and deserves to be practised.

In the event that an individual or sub-group has particular expertise on a topic, it often makes sense to delegate decisions on that topic to the experts. They will often be able to determine a clearly better outcome with little discussion. As soon as decisions have to made about trade-offs that affect other parties, however, those parties should be involved in the decision. When authority is overstepped, it breeds both resentment and poor decisions, so groups should use this strategy with care.

A team typically has a large number of decisions to make where the entire team has a certain amount of expertise. As far as I can tell, these decisions should be made using consensus when possible, democracy when necessary, and other approaches only in corner cases. Striving for consensus and democracy should occur frequently enough that they are familiar and comfortable tools. This usually means applying them to less critical decisions on a regular basis, so that the process is smooth when a critical discussion must be had.

I'm convinced that decision by consensus, when reachable, produces the best outcomes by far. When a group reaches consensus, everyone can be happy about the outcome or at least accepts the necessity of that outcome and feels like their voice mattered. Unfortunately, there is not guarantee that consensus will be reachable for a particular issue – it may be impossible or too costly to reach. When consensus fails, a good fallback is to rely on a democratic choice. There should be a clear plan how long to work on consensus, when to fallback to another process, and what process to fallback to. Usually the best fallback option is democracy.

When making a democratic choice, the usual – and usually best – approach is a poll for simple majority. When resentment might play a role, consider anonymous voting. There are many variants on the democratic decision-making process and it is worthwhile to survey some of these variants to see if improvements can be found for your process, particularly at larger scales.

Democracy is riskier than consensus, and riskier than decision by authority (when the authority has a legitimate claim to expertise). The risks of democracy include factionalization and resentment, which should be monitored and combated. In my experience, this tends to be more of a problem at large scale, and less of an issue within a small team, but if your experiences have been different please share them with me.

Putting this together, an obvious algorithm is reached:

1. Is there a clear and acknowledged expert on the topic? Then let the expert decide. Remember that experts can and should be challenged, and when the decision being made affects other parties they should be consulted and ideally part of the decision-making process.
2. In the absence of authority, go for consensus. Often decisions are pretty obvious and saying something like "It seems to me like X is a good choice, does someone object?" will steam right though since most people want to get out of the meeting room. When objections occur, let everyone have their say. You need a clear strategy to follow regarding how long you will strive for consensus, and your fallback will be when consensus fails.
3. Democracy should be the default fallback. Some people will be unhappy but, when done right, people will feel that the decision was fair, keeping morale high. In the absence of consensus or legitimate authority, this will produce the best outcome and is not costly.
4. There are rare cases where a person in a position of administrative authority, like a manager or team leader, might find it best to exercise that authority. This can be useful in extreme cases where the leader feels it is important for team morale, or has knowledge they can't share that enables them to make a more informed choice. This should only occur rarely.

Harmful decision making strategies

Doubtless there exist countless identifiable patterns of harmful group decision-making. Some like authoritarianism, monarchy, cult of personality, and fear-mongering are well understood but nonetheless continue to play a large role in both the private and public sector. We can only try to be aware of these patterns and seek to avoid them when we have the opportunity. Some other common anti-patterns I've recognized, which are perhaps less discussed, are:

- **administrative authority**: An individual is granted an administrative function which gives them the authority to override their teammates and make *decisions by fiat*. This can be very time efficient, but has many vectors by which it produces unsatisfactory outcomes. It should be an unwelcome last resort, but is sadly often the default.
- **avoidance of conflict**: Often organizations seek to find ways to avoid having discussions and conflicts among their team members. When a decision is made, it is made in the background without consulting others for fear of difficult confrontations. This typically leads to the most difficult and stubborn team members leading by default.

One common anti-pattern I've noticed in scrum is having a team leader (an administrative authority position) be the scrum-master. This organizational shortcut often leads to the default decision-making process being decision by administrative authority. When it's done well, the administrator is responsive to the input and opinions of the team, but I would argue it is better to separate the organizational concerns. My current thinking is that the administrative authority and the scrum-master should be different individuals, and the administrative authority should be consulted as the authority when it is determined that the decision to be made requires administrative expertise. When group decisions have to be made, the scrum-master can decide how long to spend trying to achieve consensus, and whether to default to authority or democracy. It should be clear to the team what the process and fallback are in advance. I think rotating the scrum-master position is also valuable.

Last words

In this article, I have formalized things which seem self-evident to me, but in my experience the logical steps indicated by these observations are rarely practised. My suggestion is for we citizens and employees to study these approaches, and seek to correct and improve whatever decision-making processes we can. Ideally, effective group decision-making will become more of an active topic of conversation as this attention can only improve our processes. I welcome any contribution a reader might wish to make to the conversation, especially if your view contrasts with mine.

I encourage all readers to seek opportunities to practise consensus-building and democracy whenever possible. Do this often enough and you may inspire others to do the same. At the very least, your colleagues will appreciate your team spirit and open-mindedness. As consensus-building and democratic choice are practised more in your organization, they should become more accepted. The positive outcomes will help to spread

One SSH Key Per Machine!

Silas S. Brown has advice on configuring secure connections.

I recently had a small nightmare when my private SSH key was compromised. I had left it on a system I trusted, but due to a mistake made by (apparently) a newly-employed system administrator, the entire contents of my home directory on that system, including the private SSH key, was made available for public download for a few hours and they didn't have logs to show whether or not it had been downloaded. So I had to quickly contact every server on which I'd ever placed my public SSH key and either change it myself or ask them to revoke it (I had to ask them in cases where the server was offline or otherwise unreachable but I was concerned it might be put back online later).

I now have a different private SSH key on each system. Instead of thinking of the private key as 'my key' that identifies me as an individual, I now think of it as identifying the system (well, my account on the system) that I'm using. I may have to add multiple public keys to each `.ssh/authorized_keys` file on the servers I have access to, but I have much more fine-grained control over which of my clients can access which servers, which should hopefully reduce the hassle next time we find a breach (if server X trusts client Y but not client Z, then I don't have to bother updating server X when I'm told client Z was compromised).

My `.ssh/authorized_keys` files now look like:

```
ssh-rsa long_key_here client1
ssh-rsa long_key_here client2
```

where **client1**, **client2** etc can be any strings I like (the third field of these files is ignored by sshd), so I simply use them to name the machines on which the corresponding private keys are stored. Each key is, of course, generated by running the **ssh-keygen** command on the corresponding client (there's no point running it elsewhere), which creates `.ssh/id_rsa` (the private key) and `.ssh/id_rsa.pub` (the public key) – I then edit the public key file and make sure the label is what I want it to be, before copying it to the `.ssh/authorized_keys` of the servers I use and/or uploading it to GitHub, GitLab and/or Bitbucket. (I did once have a problem with Bitbucket: I had both a personal account and a work account, and it wouldn't let me use the same SSH key for both, so if I wanted to access both from the same machine then I had to generate two keys on that machine and look up how to use git with ssh-agent to select an alternate key, but hopefully that's a rare situation.)

Many (but not all) SSH servers also allow you to put **from="IP"** before the ssh-rsa column, where IP is a single IP address or a list of IP addresses and/or subnets (e.g. **from="192.168.0.0/16,1.2.3.4" ssh-rsa** ...) to specify from where the client is expected to connect. This doesn't provide very much additional security, since anyone who fully compromises a client is likely to use it in-place with no change of IP address, but it may at least protect against the lesser failure mode of an attacker obtaining a copy of a private key but without actually obtaining control of the client on which it is stored. So if a particular client has a fixed IP address, or is within a fixed IP network block, it may be worth considering adding **from=** specifiers, although you will of course need to change these if the client ever does change its IP.

Listing 1

```
TCPKeepAlive yes
ServerAliveInterval 200
Compression yes
CompressionLevel 9

Host ds linux.ds.cam.ac.uk pwf
HostName linux.ds.cam.ac.uk
 User ssb22
 ControlMaster auto
 ControlPath /tmp/.dspath
```

I'm also an avid user of the `.ssh/config` file on clients to abbreviate hostnames and do other tricks. See Listing 1 for an example.

The first four options are useful over slow or unreliable links. The **Host** option lets me give the server several names (in this case including the old name it used to have, in case I ever accidentally type that one) and maps it to the real **HostName**. The **User** option makes sure I log in as the correct user even if my username on the local system is different (although I usually try to get the same username if I can). The **ControlMaster** and **ControlPath** options let me pass additional sessions through the same connection, which is useful if the server keeps your home directory on a Windows file share or something, as this often results in it requiring a password login rather than a public key login: at least **ControlPath** lets you avoid typing the password for any additional shells you want to open.

Other good options include **Port**, if the server is running on a non-standard port to reduce the amount of clutter in its logs from probes, and **ProxyCommand**, which can be set to run a 'port-knocking' script to ask the server to open its port before performing an **nc** command to it. You need to write the latter as:

```
ProxyCommand /bin/bash -c 'port-knock >/dev/null
2>/dev/null && /usr/bin/nc %h %p'
```

where **port-knock** is whatever command you use to do the 'port knocking' (or for example a curl command to run a CGI script that opens the port). ■

SILAS S. BROWN

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

A Guide to Group Decision Making (continued)

the methods. Pitfalls that have to be navigated include problematic colleagues, and excessive time-suck.

In my experience, following this advice leads to happier and more productive teams, and better decisions. As with anything worthwhile, there are challenges and intricacies to consider, but I hope you will agree that good group decision-making deserves attention and practice. ■

References

[1] Manifesto for Agile Software Development: http://agilemanifesto.org/

[2] Principles behind the Agile Manifesto: http://agilemanifesto.org/principles.html

ACCU Standards Report

Emyr Williams reports the latest from the C++ Standards meetings.

Sadly, I missed the deadline with my last report, which is entirely my fault. However, I hope to make up for it with the latest offering. The usual caveats apply. I would like to offer my everlasting thanks to Roger Orr for being kind enough to proof-read what I've written. Any errors that remain are mine alone.

First up, there's been some news about new personnel in the ISO committee, with ACCU's Nina Ranns elected to be the WG21 secretary, which is awesome, along with ACCU's Code Critique master Roger Orr, who has been invited to join the direction group, which is a small by-invitation group of experienced participants who are asked to recommend priorities for WG21. So heartiest congratulations to them both.

So, over a week in June, the C++ ISO Committee met in Rapperswil, which is in Switzerland, hosted by HSR Rapperswil, Zühlke, Netcetera, Bbv, SNV, Crealogix, Meeting C++ and BMW Car IT GmbH. It was well-attended, with 140 people at the meeting and 11 national bodies represented.

This was the penultimate meeting for merging major language features in to C++ 20, so priority was given to the larger proposals that could make C++ 20 or make solid progress towards making it in to C++ 20.

The second version of the Parallelism TS (N4725) is now finalised and, by the time this article is published, will be sent for publication by the various national bodies, including BSI, and people will be able to make use of it portably among compiler vendors that support it in their compilers.

The draft Reflection TS has been sent out to the national bodies for ballot. This is normally the final step before publication, and by the time the standards body meets again, the results of the ballot should be in and, if there are no glaring issues, they will move on towards publication.

Some of the highlights include:

Contracts adopted for C++ 20 (P0380R1)

This feature of the language provides a structured way to express function preconditions and postconditions in code.

For example, consider the following code snippet (with thanks to Jose Daniel Garcia Sanchez, the paper's author, for proof-reading my sample):

```
double deposit(int account_number,
  double deposit)
[[expects: acount_number != 0 ]]
[[ensures result: result > 0 ]]
{
  double current_balance =
    get_balance(account_number);
  // this is within a function body,
  // so it can access everything within the
  // function itself.
  [[ assert: new_balance ==
    current_balance + deposit ]];
  return new_balance;
}
```

EMYR WILLIAMS

Emyr Williams is a C++ developer who is on a mission to become a better programmer. His blog can be found at www.becomingbetter.co.uk

The contract is made up of three parts. There is the pre-condition (**expects**), the post-condition (**ensures**), and the assertion (**assert**). As can be seen from the above code sample, the post- and pre-conditions are defined outside of the function body, so they can't access the local variables within the function, other than any arguments that are passed to that function.

Contracts are also considered part of a function's interface, and they operate on the external view of the function.

In his blog, Herb Sutter gives three points as to why contracts are a big deal. I won't go in to detail in this article, but in essence they are:

1. Having decent contracts support is the first big step of reforming error handling in C++, and it will apply all the lessons learned over the last 30 years or so.
2. It will allow for the gradual migration of **std::** standard library precondition violations from exceptions or error codes to contracts. It will also help to remove a majority of all exceptions thrown by the standard library.
3. It will allow the language to consider handling out of memory differently from other errors. The LEWG voted unanimously to pursue section 4.3 of Herb's paper (P0709) which proposed a path of migrating all OOM from **bad_alloc** to **new(nothrow)**-like mechanisms.

   The initial step, not for C++20 mind you, would be to change the default **new_handler** from throwing **bad_alloc** to calling **terminate**.

It's worth noting that contracts are not permitted to perform an observable modification of an object. It's also worth adding that this is the first step, for C++20, and the expectation is that the feature will be extended in the future.

Class types in non-type template parameters (p0732)

This change has been desired for a while. This would permit template non-type parameters of user-defined types, and enables instantiation templates with values such as compile-time strings.

Turns out that this is now possible due to a side benefit of the **<=>** spaceship comparison operator, which was proposed by Herb Sutter.

This is because the semantics of a defaulted **<=>** comparison is essential, because the compiler has to perform comparisons to determine whether or not the two template instantiations are the same.

Feature test macros (p0941r2)

This allows code to portably test whether certain new features in C++ exist. Now some may wonder "What's the point of that?" Well, the plan is that it will enable teams to adopt new C++ features before the compilers you use support them.

To use them you'd write something like:

```
#if __cpp_hex_float  // feature macro's name
  // modern code here...
#else
  // do it the old way
#endif
```

Once you've moved on to the newer compiler, you can drop the **#if** test and remove the entire **#else** block. So it ensures that your code is

reasonably future proof, and backwards compatible for as long as you need it to be.

One key part of these macros is that these are *standardised* and can therefore replace a lot of the 'if gcc version > *x* or if msvc version > **y** or if clang version > *z*' tests.

Explicit(bool) p0892r2

This is a conditional explicit, along the lines as conditional **noexcept**. It lets library writers write **explicit** at a finer granularity, to turn off conversions where possible without having to write two functions at the same time.

When writing a class template which wraps a member of a template parameter type, it's useful to expose constructors that allow the user to construct the member in place.

STL concepts

The core features of concepts have already been accepted in to C++ 20, however the Evolutionary Working Group had another evening session on Concepts at the meeting in an attempt to resolve the issue of abbreviated function templates (AFTs).

The main issue was that given an AFT written using the Concepts TS syntax, such as (for example):

```
void sort(Sortable &s);
```

it's not clear that this is a template. And it assumes that you know that **Sortable** is a concept and not a type.

Two proposals were presented. Herb Sutter presented an in-place syntax proposal (P0745r1) which the previous code sample, would be written:

```
void sort (Sortable{} &s);
```

or

```
void (Sortable{S}& s);
```

The proposal also proposed to change the constrained-parameter syntax to require braces for type parameters so that you'd write instead:

```
template<Sortable{S}> void sort(S& s);
```

The second proposal was Bjarne Stroustrup's minimal solution to concept syntax problems (p1079r0), which adds a single leading template keyword to announce that an ATF is a template:

```
template void sort(Sortable& s);
```

This proposal leaves the constrained-parameter syntax alone.

Both proposals also have their downsides. Bjarne's proposal annotates the whole function rather than individual parameters so, for example, if you have a function with multiple parameters, you won't know at a glance which parameter is a concept. Herb's proposal changes the constrained-parameter syntax.

Both proposals were well received by the EWG, and there will be further discussions about this in future meetings. However, there is now a new paper in the post meeting mailing, authored by Ville Voutilainen, P1141R0 entitled 'Yet another approach for constrained declarations'. But this will be discussed in the next report.

Modules

The committee saw a merged approach, which both major proposers said satisfies their requirements, that was met with enthusiasm in the room. The merged proposal (p1103r0) aims to combine the best of the Modules TS and the Atom alternative proposal, which was approved by the EWG.

The merged proposal accomplishes Atom's goal of providing a better mechanism for existing code bases to transition to Modules via a modularised legacy header. Essentially, existing headers that are not modules but are treated as if they are modules by the compiler. The merged proposal also retains the Modules TS mechanism of global module fragments with some restrictions, such as only allowing **#include**s and other preprocessor directives in the global module fragment.

It's noteworthy, however, that the EWG didn't approve the poll to incorporate a subset of this merged proposal in to C++ 20 at this meeting. But this will be discussed in another meeting.

Graphics

At Jacksonville, the Graphics TS (p0267r8), which was set to contain 2D graphics primitives with a 'Cairo' inspired interface, ran in to some controversy. A number of folks had become convinced that, since this was something that professional graphics programmers/game developers were unlikely to use, the time required for a detailed wording review wouldn't be a good use of committee time.

As a result of this feeling, an evening session was held at the meeting to decide the future of the proposal. Initially, there was no overall consensus on whether or not the committee should proceed with ANY kind of graphics proposal.

Two papers were presented:

- Bruce Adelstein Lelbach presented the Diet Graphics paper (p1062r0), which offered a rebuttal of the need for graphics support in the language but does propose a simple importing API;
- ACCU's own Guy Davidson also gave a presentation on the current graphics paper (p0267r8) along with some code snippets, and practical demonstrations.

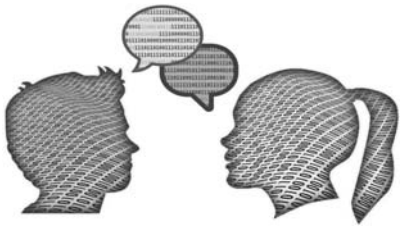
Guy also presented a paper (p0988) which detailed the history of the attempt to bring 2D graphics support into C++. Guy proposed two ways forwards: one was to kill the paper, and the other was to get confirmation that the committee still wanted to bring in 2D graphics support somehow.

This sadly led to no consensus being reached, and thus the graphics TS was abandoned for the time being.

At present it's my understanding that the BSI are pressing on with publishing the original graphics paper, so while it may not become an ISO standard for 2D graphics, it may become a BSI standard for 2D graphics.

Code Critique Competition

Set and collated by Silas Brown. 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

> Further to articles introducing D, I am attempting to use the event-driven Vibe.d server library, which I understand to be like a native version of Python's Tornado and potentially a 'killer app' of D as I haven't seen any other mature event-driven server library in a compiled language.
>
> I would like to build a back-end server that performs some processing on the body of the HTTP request it receives. To begin with, I would like it to simply echo the request body back to the client.
>
> My code works but there are three problems: (1) when I issue a POST request with lynx, 3 spaces are added to the start of the response text, (2) I cannot test with nc because it just says 404 and doesn't log anything, and (3) I'm worried that reading and writing just one byte at a time is inefficient, but I can't see how else to do it: I raised a 'more documentation needed' bug at https://github.com/vibe-d/vibe.d/issues/2139 but at the time of writing nobody has replied (should I have used a different forum?).

The code is in Listing 1.

Listing 1

```
import vibe.d;
void main() {
  auto settings = new HTTPServerSettings;
  settings.port = 8080;
  listenHTTP(settings, (req, res) {
    ubyte[1] s;
    while(!req.bodyReader.empty()) {
      req.bodyReader.read(s);
      res.bodyWriter.write(s);
    }
  });
  runApplication();
}
```

Critiques

Russel Winder <russel@winder.org.uk>

The statement of the problem gives some D code using the framework Vibe.d (a single-threaded asynchronous fibres framework with support for HTTP and HTTPS processing), but doesn't indicate how the code is to be compiled. One could, I suppose, use one of the three D compilers (dmd, ldc2, or gdc) directly, or use Meson or SCons, but you would have to have already sorted out the dependency on Vibe.d to do that. SCons has some vestigial support for handling dependencies, but it isn't in production as yet. The standard way of handling dependencies and build in the development of D projects is to use Dub. So to start, let us actually build the presented code.

It is worth noting that the original problem code is an amendment of the first example of using Vibe.d at https://github.com/vibe-d/vibe.d. That 'Hello Vibe.d' server is not really an echo, though; it is just writing a message constructed of a literal and some data from the request data structure.

It is also worth noting, that the question raised at https://github.com/vibe-d/vibe.d/issues/2139 has been answered, albeit still open. It was certainly the right thing to raise this issue.

In working on this code, I worked with Debian Sid using Emacs (obviously, though there is the D plugin for IntelliJ IDEA, which perhaps I should have used) and D 2.081.1. I used both the dmd and ldc compilers: dmd 2.081.1 was installed using packages from the D-Apt repository. ldc 1.11.0 is not yet formally released yet, so I built ldc 1.11-beta from a clone of the Git repository – it's really quite easy as long as you have a D compiler installed, Debian current has ldc 1.8.0 which is ancient, but it if fine for the job of bootstrapping the ldc build.

Dubbing it

There are two notations for creating Dub specification files: JSON and SDL. Personally I prefer SDL, even though a lot of people use JSON. See https://code.dlang.org/getting_started

There is a canonical project organisation for a D project: all the source is in the subdirectory `source`, and the entry point (the main function) of an application is either in the file `source/main.d` or `source/app.d`. Without a file with one of these names or explicit specification, the project is assumed to be a library. Using the canonical structure means an almost empty specification file when using Dub.

So with the code from original code author's statement in the file `source/main.d`, I created a `dub.sdl` file:

```
name "server"
description "A sample server using vibe.d for
CVu Code Critique 112"
dependency "vibe-d" version="*"
```

Note that I have used `"*"` as the version number for the vibe-d dependency here, which means use the latest version you can find in the Dub packages repository at http://code.dlang.org/. I could have written `"0.8.4"` to specify a specific version. The version descriptions possible are outlined at http://code.dlang.org/package-format?lang=sdl#version-specs

So with the Dub project specification file in place, all that is required to build the project is to run the command:

```
dub build
```

The result was for Dub to download the specified dependency and all transitive dependencies and compile them:

```
|> dub build --compiler=$HOME/BuildArea/LDC/bin/
ldc2
Fetching eventcore 0.8.35 (getting selected
version)...
Fetching libevent 2.0.2+2.0.16 (getting selected
version)...
Fetching diet-ng 1.5.0 (getting selected
version)...
Fetching taggedalgebraic 0.10.11 (getting
selected version)...
Fetching openssl 1.1.6+1.0.1g (getting selected
version)...
Fetching botan 1.12.10 (getting selected
version)...
```

SILAS S. BROWN

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

```
Fetching stdx-allocator 2.77.2 (getting selected
version)...
Fetching vibe-d 0.8.4 (getting selected
version)...
Fetching mir-linux-kernel 1.0.0 (getting selected
version)...
Fetching memutils 0.4.11 (getting selected
version)...
Fetching vibe-core 1.4.1 (getting selected
version)...
Fetching libasync 0.8.3 (getting selected
version)...
Fetching botan-math 1.0.3 (getting selected
version)...
Performing "debug" build using /home/users/
russel/BuildArea/LDC/bin/ldc2 for x86_64.
taggedalgebraic 0.10.11: building configuration
"library"...
eventcore 0.8.35: building configuration
"epoll"...
stdx-allocator 2.77.2: building configuration
"library"...
vibe-core 1.4.1: building configuration
"epoll"...
vibe-d:utils 0.8.4: building configuration
"library"...
vibe-d:data 0.8.4: building configuration
"library"...
mir-linux-kernel 1.0.0: building configuration
"library"...
vibe-d:crypto 0.8.4: building configuration
"library"...
diet-ng 1.5.0: building configuration
"library"...
vibe-d:stream 0.8.4: building configuration
"library"...
vibe-d:textfilter 0.8.4: building configuration
"library"...
vibe-d:inet 0.8.4: building configuration
"library"...
vibe-d:tls 0.8.4: building configuration
"openssl-1.1"...
vibe-d:http 0.8.4: building configuration
"library"...
vibe-d:mail 0.8.4: building configuration
"library"...
vibe-d:mongodb 0.8.4: building configuration
"library"...
../../../../../.dub/packages/vibe-d-0.8.4/vibe-d/
mongodb/vibe/db/mongo/settings.d(16,8):
Deprecation: alias
`std.digest.digest.toHexString` is deprecated
- import std.digest instead of std.digest.digest.
std.digest.digest will be removed in 2.084
../../../../../.dub/packages/vibe-d-0.8.4/vibe-d/
mongodb/vibe/db/mongo/settings.d(16,8):
Deprecation: alias
`std.digest.digest.toHexString` is deprecated
- import std.digest instead of std.digest.digest.
std.digest.digest will be removed in 2.084
vibe-d:redis 0.8.4: building configuration
"library"...
vibe-d:web 0.8.4: building configuration
"library"...
vibe-d 0.8.4: building configuration "vibe-
core"...
server ~master: building configuration
"application"...
```

We can ignore the deprecation warning as it is nothing to with our code, and it is a warning. Let's hope the thing gets fixed fairly soon, D 2.084.0 is not that far distant given we are on 2.081.1 now.

Note that the compiler has been explicitly specified. If **`--compiler=`**… is not specified then Dub looks for D compilers in the path using the order dmd, ldc2, gdc. I ran first with my build of ldc2 1.11-beta and show the output here. I then also ran without the **`--compile=`**… and the dmd 2.081.1 was package installed and used.

All the dependency compilation products are cached:

```
|> dub build --compiler=$HOME/BuildArea/LDC/bin/
ldc2
Performing "debug" build using /home/users/
russel/BuildArea/LDC/bin/ldc2 for x86_64.
taggedalgebraic 0.10.11: target for configuration
"library" is up to date.
eventcore 0.8.35: target for configuration
"epoll" is up to date.
stdx-allocator 2.77.2: target for configuration
"library" is up to date.
vibe-core 1.4.1: target for configuration "epoll"
is up to date.
vibe-d:utils 0.8.4: target for configuration
"library" is up to date.
vibe-d:data 0.8.4: target for configuration
"library" is up to date.
mir-linux-kernel 1.0.0: target for configuration
"library" is up to date.
vibe-d:crypto 0.8.4: target for configuration
"library" is up to date.
diet-ng 1.5.0: target for configuration "library"
is up to date.
vibe-d:stream 0.8.4: target for configuration
"library" is up to date.
vibe-d:textfilter 0.8.4: target for configuration
"library" is up to date.
vibe-d:inet 0.8.4: target for configuration
"library" is up to date.
vibe-d:tls 0.8.4: target for configuration
"openssl-1.1" is up to date.
vibe-d:http 0.8.4: target for configuration
"library" is up to date.
vibe-d:mail 0.8.4: target for configuration
"library" is up to date.
vibe-d:mongodb 0.8.4: target for configuration
"library" is up to date.
vibe-d:redis 0.8.4: target for configuration
"library" is up to date.
vibe-d:web 0.8.4: target for configuration
"library" is up to date.
vibe-d 0.8.4: target for configuration "vibe-
core" is up to date.
server ~master: target for configuration
"application" is up to date.
To force a rebuild of up-to-date targets, run
again with --force.
```

The version numbers of the dependencies currently compiled are stored in the file `dub.selections.json`. If you have not specified exact versions of the dependencies and you want to check for updates, you run **`dub upgrade`** and, if there are new versions, the dependency versions will be updated, the new versions downloaded, and then compiled.

Running it

The upshot of all the previous section is that we have a compiled program, and so we can try it out:

```
|> ./server
[main(----) INF] Listening for requests on
http://[::]:8080/
Failed to listen on 0.0.0.0:8080
```

If it isn't listening in 0.0.0.0:8080, is it listening on anything? Well the way forward is to try and see. When I navigated to http://localhost:8080 using a browser, in this case Firefox, it reports:

```
404 – Not Found

Not Found

Internal error information:
No routes match path '/'
```

which seems to indicate something is listening on 127.0.0.1:8080, but something that can't do an HTTP **GET** of **/**. The original code author did say though that they used lynx to issue a **POST**, but doesn't say what command line they used. I don't have lynx, but I do have curl on my computers, so I shall s/lynx/curl/ and try an experiment.

```
|> curl --data-binary "Hello World"
http://localhost:8080
Hello World
```

OK, so the original code author's claim is confirmed, it is echoing HTTP **POST** data. Phew. ☺

Except, I am not seeing three leading spaces added to the return. Should I smell a rat or is this perhaps a 'Lynx Effect'?

As for reading and writing a single byte at a time, the person answering the original code author's issue at https://github.com/vibe-d/vibe.d/issues/2139 states that a canonical loop should look like:

```
while (!conn.empty) {
  ubyte[256] buf = void;
  auto n = conn.read(buf[], IOMode.once);
  dst.write(buf[0 .. n]);
}
```

so I thought I should try that.

Updated version

Well, except, I didn't. The **conn** and **dst** in the code fragment raised a 'flag'. The delegate (aka lambda function) that is the function passed into the **listenHTTP** function in the original code is written to take parameters **req** and **res** of inferred type. This seems like HTTP request data and HTTP result data, which doesn't seem like a TCP connection type thing as inferred from the name **conn**.

Delving into the examples from the Vibe.d GitHub repository, we find at https://github.com/vibe-d/vibe.d/blob/master/examples/echoserver/source/app.d:

```
import vibe.appmain;
import vibe.core.net;
import vibe.core.stream;

shared static this()
{
  listenTCP(2000, (conn) {
    try conn.pipe(conn);
    catch (Exception e) conn.close();
  });
}
```

which I think we can assume is the canonical echo server written in D using Vibe.d. Notice it uses **pipe**, a function that is mentioned in the reply at https://github.com/vibe-d/vibe.d/issues/2139.

Interesting code. The **shared static this()** is the module initialisation block, executed when the executable starts, before **main** is executed. Why no **main** function in the source? Because it is boilerplate. **import vibe.appmain** imports the needed **main**.

NB For C and C++ folk: there are no **#include** statements because D is a module-based language. No textual code inclusion, just access to other modules via import statements. Most modern. ☺

Obviously, though, this is a TCP server, not an HTTP server. If the goal was to have a packet echo server then 'Job Done'. However, the original code author emphasised that HTTP **POST**s were expected.

POSTing version

Let us look again at the examples from the Vibe.d GitHub repository. We immediately find an HTTP server (https://github.com/vibe-d/vibe.d/blob/master/examples/http_server/source/app.d) and an HTTPS server (https://github.com/vibe-d/vibe.d/blob/master/examples/https_server/source/app.d). Clearly, we should only be considering HTTPS for production code. However, as can be seen from the two example listings, an HTTPS server is an HTTP server with added settings. So, for experimentation we can simplify by working with HTTP, but then ensure we add the extra settings and keys to use HTTPS before going into production.

So, based on the HTTP server example (so it can be tried without creating keys etc.) here is a bit of code that responds to **GET**s with a message so as to be certain the right server has been reached and that echoes **POST** message bodies.

```
import vibe.appmain;
import vibe.http.server;

void handleRequest(scope HTTPServerRequest req,
scope HTTPServerResponse res) {
  if (req.method == HTTPMethod.POST) {
    res.writeBody(req.bodyReader.peek(),
      "text/plain");
  } else {
    res.writeBody("I got a GET.\n",
      "text/plain");
  }
}

shared static this() {
  auto settings = new HTTPServerSettings;
  settings.port = 8080;
  settings.bindAddresses = ["::1",
    "127.0.0.1"];
  listenHTTP(settings, &handleRequest);
}
```

This I would claim is fairly canonical Vibe.d use. Use the default Vibe.d **main**, set up all the server setting in the module initialisation, with a top level function that is the callback for all server requests.

In the handler, control is split for **POST** and **GET** requests. In the **GET** request branch we just write a valid response using methods provided by the **HTTPServerResponse** instance **res**. In the **POST** request branch, we get a reference to the buffer in the **HTTPServerRequest** instance **req** that holds the **POST**ed data and construct a response using that. Seems like the very essence of echo.

Comparing this to the original code, and perhaps explaining why it was wrong, and this version at least better: the crux of this version is using:

```
HTTPServerResponse.writeBody
```

instead of:

```
HTTPServerResponse.bodyWriter.write
```

and:

```
HTTPServerRequest.bodyReader.peek
```

to access the request body rather than using

```
HTTPServerRequest.bodyReader.read
```

to read it byte at a time. By using the higher level functions, any looping is implicit, so no explicit **while** statement – more declarative.

To compile this version and show it meets the requirements, we must add a line to the `dub.sdl` file so that it reads:

```
name "server"
description "A sample server using vibe.d for
CVu Code Critique 112"
dependency "vibe-d" version="*"

versions "VibeDefaultMain"
```

The default **main** system requires the use of conditional compilation D style. So we set the compilation version to **VibeDefaultMain** and now everything compiles and indeed runs:

```
|> dub run --compiler=$HOME/BuildArea/LDC/bin/
ldc2
Performing "debug" build using /home/users/
russel/BuildArea/LDC/bin/ldc2 for x86_64.
taggedalgebraic 0.10.11: building configuration
"library"...
eventcore 0.8.35: building configuration
"epoll"...
stdx-allocator 2.77.2: building configuration
"library"...
vibe-core 1.4.1: building configuration
"epoll"...
vibe-d:utils 0.8.4: building configuration
"library"...
vibe-d:data 0.8.4: building configuration
"library"...
mir-linux-kernel 1.0.0: building configuration
"library"...
vibe-d:crypto 0.8.4: building configuration
"library"...
diet-ng 1.5.0: building configuration
"library"...
vibe-d:stream 0.8.4: building configuration
"library"...
vibe-d:textfilter 0.8.4: building configuration
"library"...
vibe-d:inet 0.8.4: building configuration
"library"...
vibe-d:tls 0.8.4: building configuration
"openssl-1.1"...
vibe-d:http 0.8.4: building configuration
"library"...
vibe-d:mail 0.8.4: building configuration
"library"...
vibe-d:mongodb 0.8.4: building configuration
"library"...
../../../../../.dub/packages/vibe-d-0.8.4/vibe-d/
mongodb/vibe/db/mongo/settings.d(16,8):
Deprecation: alias
`std.digest.digest.toHexString` is deprecated -
import std.digest instead of std.digest.digest.
std.digest.digest will be removed in 2.084
../../../../../.dub/packages/vibe-d-0.8.4/vibe-d/
mongodb/vibe/db/mongo/settings.d(16,8):
Deprecation: alias
`std.digest.digest.toHexString` is deprecated -
import std.digest instead of std.digest.digest.
std.digest.digest will be removed in 2.084
vibe-d:redis 0.8.4: building configuration
"library"...
vibe-d:web 0.8.4: building configuration
"library"...
vibe-d 0.8.4: building configuration "vibe-
core"...
server ~master: building configuration
"application"...
Running ./server
[main(----) INF] Listening for requests on
http://[::1]:8080/
[main(----) INF] Listening for requests on
http://127.0.0.1:8080/
```

and now if we curl things in:

```
|> curl http://localhost:8080
I got a GET.
|> curl --data-binary "Hello World"
http://localhost:8080
Hello World
```

Is it now time to say 'Job Done'? Probably.

Addendum

Having submitted my response to Code Critique 112 to the appropriate authorities, it seems the original problem setter reviewed the piece and came up with a problem. From the email:

> Unfortunately, though, his revised **POST**-handling code works with small requests but not with large ones. On my system:
>
> ```
> curl --data-binary $(yes | head -300)
> http://localhost:8080 | wc -l
> 299
> ```
>
> (this is OK: the 300th line has no **\n** terminator as per HTTP **POST** semantics), but
>
> ```
> curl --data-binary "$(yes|head -3000)"
> http://localhost:8080 | wc -l
> 0
> ```
>
> (this is not OK.) I think the reason for this is the reliance on **peek()**, the documentation of which says "Returns a temporary reference to the data that is currently buffered". There is no guarantee that ALL the data will currently be buffered, especially if there's a lot of it.

This is not only entirely correct, it highlights that I failed to do proper system testing. I shall, of course, take myself immediately off into a corner and tear strips off myself until I am appropriately contrite.

Some minutes later:

So, first I must reproduce the error (see Figure 1).

So now to deal with the problem of **peek**. That algorithm made the, clearly unreasonable, assumption that data would always come in a single buffer; without delving into the implementation, it is not clear what the exact problem is, but we know using **peek** doesn't work as tried in the submitted code.

Original problem setter goes on to say:

> The answer on Vibe.d issue 2139 said that a canonical **read** loop would be:
>
> ```
> while (!conn.empty) {
>   ubyte[256] buf = void;
>   auto n = conn.read(buf[], IOMode.once);
>   dst.write(buf[0 .. n]);
> }
> ```

so I tried changing the **HTTPMethod.POST** branch to:

```
|> curl --data-binary "$(yes | head -300)" http://localhost:8080 | wc -l
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  1198  100   599  100   599  42785  42785 --:--:-- --:--:-- --:--:-- 85571
299

|> curl --data-binary "$(yes | head -3000)" http://localhost:8080 | wc -l
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  5999    0     0  100  5999      0   976k --:--:-- --:--:-- --:--:--  976k
0
```

Figure 1

```
ubyte[] rBody;
while (!req.bodyReader.empty) {
  ubyte[256] buf = void;
  auto n = req.bodyReader.read(buf[],
    IOMode.once);
 rBody ~= buf[0 .. n];
}
// (optionally do processing on rBody here)
res.writeBody(rBody, "text/plain");
```

and adding `import vibe.d;` at the start, but the result was the message "413 – Request Entity Too Large" even with just a single-byte argument in curl's `--data-binary` option. This doesn't seem to make much sense: why would it let you see (small) requests when you `peek()`, but send the client an error when you `read()`? I suppose we will have to wait for Vibe.d to add more beginner-friendly documentation.

So, I should test this updated code (see Figure 2 for the results):

```
import vibe.appmain;
import vibe.http.server;
import eventcore.driver: IOMode;

void handleRequest(scope HTTPServerRequest req,
   scope HTTPServerResponse res) {
  if (req.method == HTTPMethod.POST) {
    ubyte[] rBody;
    while (!req.bodyReader.empty) {
      ubyte[256] buf = void;
      auto n = req.bodyReader.read(
        buf, IOMode.once);
      rBody ~= buf[0 .. n];
    }
    res.writeBody(rBody, "text/plain");
  } else {
    res.writeBody("I got a GET.\n",
      "text/plain");
  }
}
shared static this() {
  auto settings = new HTTPServerSettings;
  settings.port = 8080;
  settings.bindAddresses = ["::1",
    "127.0.0.1"];
  listenHTTP(settings, &handleRequest);
}
```

Not the result original problem setter got, but still wrong. In fact, the original problem setter's result is obtained if the `| wc -l` is removed. Obvious really, but it took me a moment:

```
|> curl --data-binary "$(yes | head -300)"
http://localhost:8080
413 - Request Entity Too Large

Request Entity Too Large
|> curl --data-binary "$(yes | head -3000)"
http://localhost:8080
413 - Request Entity Too Large

Request Entity Too Large
```

It seemed likely that this is the result of an exception, probably in the `req.bodyReader.read`. Experimenting with a `try`/`catch` block indicated this was indeed the case. So it seems there is some tension here between `req.bodyReader.empty` being false and yet `req.bodyReader.read` having a problem.

Despite the manual entry https://vibed.org/api/vibe.core.stream/InputStream.read, it would seem that this function always tries to read to fill all of the buffer and if the stream ends before the read is complete an exception is thrown. One could argue, given the signature of the function, that this behaviour is wrong, and indeed broken: the `read` should happen until the end of stream and the number of items read returned.

Anger.

I posted a message to the D Learn email list and put a post on the vibe.d forum, even though I hate forums. A most splendid person on the D Learn email list took up my case and checked the source code. It seems that the code does not implement the documentation; the behaviour I saw is the expected behaviour given the source code. However, this person also said that no active vibe.d user would be using these methods so, in a sense, the bug isn't all that surprising because no-one is exercising them. It does, though, imply the vibe.d tests are a bit short of coverage of public API.

The person did also assure me that there are higher level ways of working and that is what all vibe.d users would actually be doing, that these all work exactly as documented and, indeed, as desired. I shall take it upon myself to try and get that person to write a follow up article to this Code Critique to be published in *CVu*.

A bit more anger.

Some hours later:

So clearly I have to take vibe.d as it is, not as it is documented. In this spirit, I have to:

- catch the exception indicating we have got to the end of the stream representing the request body,
- find out how many `ubyte`s are left to read,
- create an appropriate size buffer,
- read the remaining `ubyte`s,
- add these to the response body.

So now I have:

```
import vibe.appmain;
import vibe.http.server;
import eventcore.driver: IOMode;

void handleRequest(scope HTTPServerRequest req,
   scope HTTPServerResponse res) {
  if (req.method == HTTPMethod.POST) {
    ubyte[] resBody;
    enum bufferSize = 256;
    while (!req.bodyReader.empty) {
```

Figure 2

```
|> curl --data-binary "$(yes | head -300)" http://localhost:8080 | wc -l
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   655  100    56  100   599   9333  99833 --:--:-- --:--:-- --:--:--  106k
2

|> curl --data-binary "$(yes | head -3000)" http://localhost:8080 | wc -l
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  6055  100    56  100  5999  11200  1171k --:--:-- --:--:-- --:--:-- 1182k
2

|> ll
```

```
        ubyte[bufferSize] buffer;
        // req.bodyReader.read does not, as
        // of 0.8.4, behave as documented.
        // The function either completely
        // fills the buffer or throws an
        // exception.
        try {
          auto n = req.bodyReader.read(
            buffer, IOMode.once);
          assert(n == bufferSize);
          resBody ~= buffer[0 .. $];
        } catch (Throwable t) {
          // There were not 256 items to
          // read, so lets use a deprecated
          // method to get what we need. No
          // idea what we'll be able to do
          // after this property gets
          // removed.
          auto count =
            req.bodyReader.leastSize;
          assert(count < bufferSize);
          ubyte[] bufferSlice =
            buffer[0 .. count];
          auto n = req.bodyReader.read(
            bufferSlice, IOMode.once);
          assert(n == count);
          resBody ~= bufferSlice[0 .. $];
          break;
        }
      }
      res.writeBody(resBody, "text/plain");
    } else {
      res.writeBody("I got a GET.\n",
        "text/plain");
    }
  }
  shared static this() {
    auto settings = new HTTPServerSettings;
    settings.port = 8080;
    settings.bindAddresses = ["::1",
      "127.0.0.1"];
    listenHTTP(settings, &handleRequest);
  }
```

Of course, the first thing has to be to test this with original problem setters test (see Figure 3).

Exciting, this seems to be the right result. Let me test the earlier test to make sure that hasn't broken:

```
|> curl http://localhost:8080
I got a GET.

|> curl --data-binary "Hello World"
http://  localhost:8080
Hello World
```

Am I done? Well in one sense yes, I dealt with original problem setters problem with the previously submitted code, in the process uncovering something not entirely good with a bit of the vibe.d API. However, the code is a hack, it shouldn't have to be like this.

If I have to use deprecated features anyway to solve the problem, why not go for it and put the deprecated feature at the heart of the code now so as to make the code as uncomplicated as possible. Certainly this leaves the problem of dealing with the lack of feature when it is removed, but when that happens maybe the API will be different in the light of bugs and lobbying?

So the plan is:

- remove the fixed size buffer, use dynamically allocated buffers always,
- arrange that the call to req.bodyReader.read so it always reads as much as it can and no more, no less,
- remove the try/catch since there should never be a read shorter than the buffer.

This leads to the code:

```
import vibe.appmain;
import vibe.http.server;
import eventcore.driver: IOMode;
void handleRequest(scope HTTPServerRequest req,
    scope HTTPServerResponse res) {
  if (req.method == HTTPMethod.POST) {
    ubyte[] resBody;
    while (!req.bodyReader.empty) {
    //NB The leastSize property is deprecated.:-(
      auto buffer = new
        ubyte[req.bodyReader.leastSize];
      auto n = req.bodyReader.read(
        buffer, IOMode.once);
      resBody ~= buffer[0..$];
    }
    res.writeBody(resBody, "text/plain");
  } else {
    res.writeBody("I got a GET.\n",
     "text/plain");
  }
}
shared static this() {
  auto settings = new HTTPServerSettings;
  settings.port = 8080;
  settings.bindAddresses = ["::1",
    "127.0.0.1"];
  listenHTTP(settings, &handleRequest);
}
```

Does it pass the tests I have? (See Figure 4 to find out.)

```
|> curl http://localhost:8080
I got a GET.

|> curl --data-binary "Hello World"
http://localhost:8080
Hello World
```

Seems like a yes. The code seems very much nicer expressed like this. It's just that it relies on deprecated features. I think this presages a lobbying

Figure 3

```
|> curl --data-binary "$(yes | head -300)" http://localhost:8080 | wc -l
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  1198  100   599  100   599   292k   292k --:--:-- --:--:-- --:--:--  584k
299

|> curl --data-binary "$(yes | head -3000)" http://localhost:8080 | wc -l
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 11998  100  5999  100  5999  2929k  2929k --:--:-- --:--:-- --:--:-- 5858k
2999
```

Figure 4

```
|> curl --data-binary "$(yes | head -300)" http://localhost:8080 | wc -l
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  1198  100   599  100   599   116k   116k --:--:-- --:--:-- --:--:--  233k
299

|> curl --data-binary "$(yes | head -3000)" http://localhost:8080 | wc -l
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 13901  100  7902  100  5999  1543k  1171k --:--:-- --:--:-- --:--:-- 2715k
2999
```

campaign to find out why the feature is deprecated and to reverse that decision before the feature actually gets removed.

Obviously, it may be that a vibe.d expert could tell us why this solution is bad and what the right solution should be.

[Editor's note: Russel goes on to say that Sönke (the leader of the vibe.d project) has written on the forum that the behaviour of **`read(buffer, IOMode)`** we have seen does indeed seem like a bug. It is going to be investigated. However, it turns out not to be a trivial thing to fix because of some things. Russel has posted an issue on all this on GitHub:

https://github.com/vibe-d/vibe.d/issues/2194]

Jason Spencer <contact@jasonspencer.org>

This is my first foray into D, but I'll give it a go…

Firstly, I absolutely agree with 'killer app' – because the different compiler versions, package versions, too-clever-for-their-own-good build systems may well end up killing somebody. They tried with me.

Addressing your problems in order:

Post with Lynx

The reason for the three spaces is that Lynx is an interactive browser and attempts to format the response in a more readable way. To disable this and just see the unformatted response, use the **`-source`** switch (see Figure 5).

Alternatively use wget [1] or curl [2] for command line HTTP clients.

404 with nc

The thing to note about nc is that it's not an HTTP client but a command line tool for creating raw TCP or UDP sockets (or listening sockets). So we have to send the correct HTTP request with all the headers and the correct delimitation. Without seeing the original nc command, it's hard to know why the response was 404.

Using Hobbit's original netcat [3] something like this works:

```
$ echo -en "POST / HTTP/1.1\r\nUser-Agent: CVu
FTW\r\nAccept: \*/\*\r\nAccept-Encoding:
identity\r\nHost: 127.0.0.1:8080\r\nContent-
Length:6\r\n\r\nWIBBLE" | nc -i 3 -v 127.0.0.1
8080
Connection to 127.0.0.1 8080 port [tcp/http-alt]
succeeded!
HTTP/1.1 200 OK
Server: vibe.d/1.4.1
Date: Tue, 31 Jul 2018 21:00:07 GMT
Keep-Alive: timeout=10
Transfer-Encoding: chunked
6
WIBBLE
0
```

Things to note here are that we're using a **`POST`** command. Also, we're using CR-LF (**`/r/n`**) line endings. The strange **`6`** and **`0`** in the response are because by default vibe.d sends chunked responses (as per the Transfer-Encoding header), and for good reason. The **`6`** and the **`0`** are both the number of bytes about to be sent. The **`0`** indicates the end of output. Chunking is often used when it's not known how much data is to be sent – and since vibe.d doesn't buffer the entire response before sending it, at least when you're using streams, it doesn't know the size of the whole response. To write the response in one go, try:

```
HTTPServerResponse.writeBody.
```

Echoing efficiency

It is hard to understand what the program is supposed to do exactly – is it to reply to **`GET`** requests? In which case, why read the body? There is no body in a GET request. If it is the student's intention to echo the headers, **`req.bodyReader`** will only access the contents of the body, and not the headers. The headers can be read through the **`req.headers`** dictionary.

There are at least two ways to make this more efficient, though. Firstly, the array can be made bigger, say 65536 bytes, and a call to **`InputStream.read ( scope ubyte[] dst, IOMode mode )`** will return the number of bytes actually read into **`dst`** (there could be less than 65536 bytes available). You can then write this many to the response output stream. You can manipulate the body data before sending it, but beware of encoding types and that the data will be presented in fixed blocks, and not line delimited, for example.

Alternatively, if all we want to do is echo back the **`POST`**ed body, we can link the request body input stream with the response body output stream with a pipe:

```
import std.stdio;
import vibe.d;

shared static this()
{
  auto settings = new HTTPServerSettings;
  settings.port = 8080;
  listenHTTP(settings, (req, res) {
    stdout.writef( "Got %d request type from %s\n",
    req.method, req.clientAddress.toString() );
    auto N = pipe(req.bodyReader,res.bodyWriter);
    stdout.writef( "piped %d bytes\n", N);
  });
}
```

Figure 5

```
$ echo -n "WIBBLE" | lynx http://127.0.0.1:8080 --post_data|hexdump -C
00000000  20 20 20 57 49 42 42 4c  45 0a 0a                 |   WIBBLE..|0000000b
$ echo -n "WIBBLE" | lynx http://127.0.0.1:8080 --post_data -source|hexdump -C
00000000  57 49 42 42 4c 45                                 |WIBBLE|00000006
```

The echoing of the body only really makes sense when the request type is a **POST**, but for debug purposes I'm printing the type to the console. This value can be checked against the **vibe.http.common.HTTPMethod** Enum to see which type it is.

Also, consider including a Content-Type header to match that of the request – if vibe.d assumed the type, and the request is a different type, then the client making the request will just get confused. The header can be accessed through **HTTPServerRequest.contentType**.

Note that I've switched from using **main()** to **this()** as newer vibe.d versions provide a **main** by default and we just need to state what we want the application to do.

I think that's all, without a better idea of what the code was supposed to do. The problems were mostly actually in understanding HTTP and the various tools. [4] From what I've seen, Vibe.d works ok, but still has a few rough edges, and the documentation, as mentioned, is not great.

In regard to the student's comment about a mature event-driven server library in a compiled language, perhaps it's not been around as long as vibe.d but there is Boost Beast [5] for a C++ equivalent.

References

[1] https://www.gnu.org/software/wget/
[2] https://curl.haxx.se/
[3] http://nc110.sourceforge.net/
[4] https://tools.ietf.org/html/rfc2616
[5] http://www.boost.org/libs/beast

Commentary

Jason was able to address the Lynx issue, but he didn't quite understand the original problem. That's partly my fault for not having stated it more clearly. The key sentence was "I would like to write a back-end server that performs some processing on the body of the HTTP request it receives." The phrase "body of the HTTP request" means it must be a **POST** request, and the "echo the request body back to the client" goal was just a 'get it up and running' prelude to adding the extra processing. Perhaps I should have given an example: "I want to write a back-end server that takes a POST request of some English text and returns a translation into French, but, before I start on that massively clever machine-translation function, I'd like to return the English as-is, just to get the basic server infrastructure working."

From this it can be inferred that **pipe()** is not really an adequate solution because, although it technically does what was asked for, it doesn't have an obvious hook where I can add my "text = my_clever_translation_function(text)" later. I do, however, appreciate Jason's suggestion of looking at Vinnie Falco's Boost.Beast C++ library – Boost.Beast does seem to require a LOT more user code than Vibe.d, but at least Boost.Beast currently seems to be better documented with good examples to get you going.

Russel was able to get his submission in early and look into fixing the problem that came up with the use of **peek()**. (No need to tear strips off himself though: we all make mistakes!) His use of the D Learn email list and Vibe.d forum is a particularly good example for a beginner because it shows where more help can be found. It's a pity that Vibe.d does not work as documented and he ended up having to use deprecated features, but it's probably the best thing that could be done within the time constraints, and it resulted in issues being opened and looked at, which has got to be a positive outcome.

Meanwhile I decided not to rewrite my Python Tornadoweb-based server into Vibe.d (unless a really good forthcoming article changes my mind); instead I improved the existing server's responsiveness by using Python's **ctypes** library to access a string-processing function I had written in C. Before using ctypes, the best I could do was shell out to the command line to pipe text through my C program on a per-request basis, which meant there was a process-launch delay for every request. But when I used ctypes, that function became part of a shared library that was preloaded into the Python process, eliminating the per-request load delay.

The Winner of CC 112

It has to be Russel, for his engagement with (and prompting improvement in) the D community, and for coming up with code that can clearly be used as a starting point for arbitrary processing of the submitted text. Jason's effort deserves an honourable mention, though.

Code Critique 113

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

This is a very short code sample but nonetheless an interesting problem.

The writer is using a variadic template to pass multiple arguments to the **least** function, which uses **std::sort** to find the smallest input number. They report the code used to work in 32-bit on a couple of compilers, but it's not reliable when used in more modern 64-bit projects.

Can you identify the problem(s), and suggest any fixes (or alternative approaches)?

The code is in Listing 2.

You can also get the current problem from the accu-general mail list (next entry is posted around the last issue's deadline) or from the ACCU website (http://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

```
#include <algorithm>
#include <iostream>

// find the least value in the arguments
template <typename T, typename... Ts>
T least(T first, Ts... rest)
{
  std::sort(&first,
    &first + 1 + sizeof...(rest));
  return first;
}

int main()
{
  std::cout << "least(1): "
    << least(1) << std::endl;
  std::cout << "least(1,2,3,4): "
    << least(1,2,3,4) << std::endl;
  std::cout << "least(10,9,8,7,6,5,4,3,2,1): "
    << least(10,9,8,7,6,5,4,3,2,1)
    << std::endl;
}
```

Challenge 4 Report & Outlining Challenge 5

Francis Glassborow presents the responses to the challenge from last time and outlines the next one.

The challenge from the last *C Vu* was to write a program that outputs the numbers from 1 to 100 without using **if**, **for**, **switch**, or **while**. I forgot to exclude the ternary operator. However, considerable ingenuity was exhibited by all. To save space and repetition, I have omitted solutions using the ternary operator unless they have some other ingenuity.

Before I get on with the submissions, here is an idea for a C++ solution which has not been used in any of the (often very ingenious) solutions.

Use a class constructor:

```
struct challenge4 {
 static int counter =0;
 static int incr;
 challenge4(){
   cout << counter << \n;
   counter += incr;
};
```

Now you can set the counter to the start value and create a vector of **(end-start -1)** to output the numbers from start to end. The value passed to **incr** can be calculated with:

```
Challenge4::incr =(end-start)/abs(end-start);
```

Or using **signbit()**:

```
int direction[] = {-1, 1}
challenge4::incr = direction[signbit(start-end)]
```

Now executing something like

```
vector<Challenge4> c4(abs(end-start -1);
```

I leave it as an exercise for the reader to take those ideas into a working program. You will need to ensure you understand how exactly how C++ initialises containers to get this right as I have deliberately written buggy code snippets. For some of you, the faults will stand out immediately; but some of you may not be quite so familiar with the finer details and have to think or even look up details in your reference sources.

Now to the readers efforts.

From Silas

Francis: Sometimes the path from a first cut to the final solution is instructive.

I was sorry to read about the lack of response to Challenge 3.

I knew those things were called 'quines' and thought plenty of members would look up how to make them, but it's easy to forget that not everyone knows such things.

For challenge 4 (outputting numbers **first** to **last** without using **if**, **for**, **switch** or **while**), a very dirty way in C is simply to use **goto** and the **?:** operator:

```
#include <stdio.h>
#include <stdlib.h>
enum { first=1, last=100 };
int main() {
  int c = first;
loop:
  printf("%d\n", c);
  (c==last) ? exit(0) : 0;
  c += (last >= first) ? 1 : -1;
  goto loop;
}
```

The fact that **exit()**, a function with side effects, can be included in a **?:** expression means we can terminate without an **if**. Of course, this won't do if we additionally say that the function is to be included in a larger program and should therefore return to its caller rather than calling **exit()**. But there's another standard function (thankfully little used) which can transfer control back to the caller: **longjmp**. Here's the **longjmp** version (and for variety let's do it via recursion rather than a **goto**) :

```
#include <stdio.h>
#include <setjmp.h>
enum { first=1, last=100 };
int loop(int c, jmp_buf e) {
  printf("%d\n", c);
  (c==last) ? longjmp(e,1) : 0;
  loop (c + ((last >= first) ? 1 : -1), e);
}
int main() {
  jmp_buf e;
  setjmp(e) ? 0 : loop(first, e);
}
```

What if we additionally ban the **?:** operator? In that case we can rewrite our loop function like this:

```
int finished(jmp_buf e) {
  longjmp(e,1);
}
int loop(int c, jmp_buf e) {
  printf("%d\n", c);
  c==last && finished(e);
  loop (c + (last >= first) - (last < first), e);
}
```

It's necessary to put the **longjmp** in a different function because it returns **void**, so the compiler won't allow us just to say **c==last && longjmp(e,1);** and the same goes for **exit()**.

And if we also ban the use of **longjmp**, we can instead write it like this:

```
#include <stdio.h>
enum { first=1, last=100 };
int loop(int c) {
  printf("%d\n", c);
  c != last && loop (c + (last > c) -
    (last < c));
}
int main() { loop(first); }
```

which, come to think of it, is probably what I should have written to begin with.

Francis: But the scenic tour is illuminating.

From James Holland

Francis: This one explores various C++ solutions showing considerable ingenuity but misses the 'aha' of populating a container using a type with suitable constructors.

FRANCIS GLASSBOROW

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

Francis's challenge deprives us of most flow control constructs. There are, however, one or two left that could still be used. As Francis implies, the challenge is probably easier to solve using C++ rather than C. My first attempt employs the conditional expression (**?:**) and recursion as shown below.

```
#include <iostream>
void increment(int amount, int value, int last)
{
  std::cout << value << ' ';
  value - last ? increment(amount,
    value + amount, last) : static_cast<void>(0);
}

int main(int argc, const char * argv[])
{
  const int first = std::stoi(argv[1]);
  const int last = std::stoi(argv[2]);
  increment(first > last ? -1 : 1, first, last);
  std::cout << '\n';
}
```

This solution prints the number in the correct order (beginning with **first**) and copes with negative values of **first** and **last**. It would be better, however, not to use a conditional expression if possible. Another feature of C++ that controls program flow is exceptions. If exceptions are to be used, something needs to be found that generates an exception when supplied with a value that signifies the task of printing the numbers is complete. It would be helpful if such an object was part of the standard library. There are probably several such objects. The one I have chosen is **std::bitset** as it has a member function that throws an exception when the value of its parameter is not an index of one of its bits. I have used this feature in the code below.

```
#include <bitset>
#include <iostream>
void increment(int amount, int value, int last)
{
  std::cout << value << ' ';
  std::bitset<1> bs;
  try
  {
    bs.reset(value - last);
  }
  catch (...)
  {
    increment(amount, value + amount, last);
  }
}

int main(int argc, const char * argv[])
{
  const int first = std::stoi(argv[1]);
  const int last = std::stoi(argv[2]);
  const int amount = 1 - 2 * (first > last);
  increment(amount, first, last);
  std::cout << '\n';
}
```

The **std::bitset** object **bs** contains just one bit and, therefore, has an index of zero. While the sequence is being written, **value - last** will not be a valid index and so **bs** will throw an exception. The exception is immediately caught causing **increment()** to be called recursively. Just after the last number is printed, **value - last** will equal 0. This is a valid index of **bs**. No exception is thrown and **increment()** recursively returns to **main()** where, after a final line-feed, the program terminates.

Also in the above program, instead of using the conditional expression to determine the value (1 or -1) by which the sequence should increment, I have constructed a little formula to do the job. Firstly, **(first > last)** is implicitly converted to 1 if **first** is greater than **last**, 0 otherwise. Multiplying this value by -2 gives -2 and 0 for the two conditions. Finally, adding 1 gives 1 or -1 as required.

The **std::bitset** technique can also be used to iterate instead of recurse as shown below.

```
#include <bitset>
#include <iostream>

int main(int argc, const char * argv[])
{
  const int first = std::stoi(argv[1]);
  const int last = std::stoi(argv[2]);
  const int amount = 1 - 2 * (first > last);
  int i = first;
loop:
  std::cout << i << ' ';
  try
  {
    std::bitset<1> bs;
    bs.reset(i == last);
    i += amount;
    goto loop;
  }
  catch (...)
  {
    std::cout << '\n';
  }
}
```

The program uses a **goto** statement that may be frowned upon. Is there a way of meeting the challenge without using iteration or recursion thus getting rid of explicit flow control altogether? Well, I think there is. The main feature of the program shown below is its use of **std::iota()**.

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <numeric>
#include <vector>

int main(int argc, const char * argv[])
{
  const int first = std::stoi(argv[1]);
  const int last = std::stoi(argv[2]);
  const int length = std::abs(first - last) + 1;
  std::vector<int> numbers(length);
  std::iota(numbers.begin(), numbers.end(), 0);
  const int m = 1 - 2 * (first > last);
  const auto f = [m, first](int x) {
    return m * x + first;};
  const std::ostream_iterator<int>
    output(std::cout, " ");
  std::transform(numbers.cbegin(),
    numbers.cend(), output, f);
  std::cout << '\n';
}
```

An **std::vector** is created that contains a sequence of values from zero to one less that the number of values to be printed. The coefficients of the formula designed to map the values in the vector to the required output sequence is then calculated. This is based on the equation $y = mx + c$, where y is the desired printed value and m is the value of the corresponding element in the vector. Within the program, c has the value of **last**. The function **std::transform()** is used to successively convert to the required value each element of the vector and then print it.

As can be seen from the above discussion, C++ provides sufficient flexibility to provide various solutions to the challenge. The C language is not quite so accommodating.

Using the conditional operator, a C language solution is not too difficult. A recursive version is shown below.

```
#include <stdio.h>
#include <stdlib.h>
void increment(int amount, int value, int last)
{
```

```
  printf("%d ", value);
  value != last ? increment(amount,
    value + amount, last) : (void)0;
}
int main(int argc, const char * argv[])
{
  const int first = atol(argv[1]);
  const int last = atol(argv[2]);
  increment(first > last ? -1 : 1, first, last);
  putchar('\n');
}
```

Alternatively, an iterative version can be provided.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, const char * argv[])
{
  const int first = atol(argv[1]);
  const int last = atol(argv[2]);
  const int amount = 1 - 2 * (first > last);
  int i = first;
loop:
  printf("%d ", i);
  (i == last) ? exit(0) : (void)0;
  i += amount;
  goto loop;
}
```

Providing a C language solution that does not use a conditional expression is a little more difficult. The best I can manage is shown below.

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, const char * argv[])
{
  const int first = atol(argv[1]);
  const int last = atol(argv[2]);
  const int amount = 1 - 2 * (first > last);
  int i = first;
loop:
  printf("%d ", i);
  assert(i != last);
  i += amount;
  goto loop;
}
```

The program certainly prints the numbers from **first** to **last** but other unwanted text is also printed due to the **assert** statement. I am not familiar with the later C standards and better solutions may be possible.

From Paul Davies

I have these entries for Francis. He didn't give an email so no wonder he didn't get many responses

Francis: Excellent point, please send submissions to francis.glassborow@gmail.com

The following is a c-quine which is 'portable'. I based it on a C++ one I wrote in 2006 and just changed the **cout**s into **printf**.

```
#include<stdio.h>
char p[]="\";int main(){
  printf(\"#include<stdio.h>\\nchar p[]=\\\"\");
  for (char* q=p;*q;++q){
    if(*q=='\\\\'||*q=='\"')printf(\"\\\\\");
    printf(\"%c\",*q);}
  printf(\"%s\\n\",p);
  return 0;}";
  int main(){
    printf("#include<stdio.h>\nchar p[]=\"");
    for (char* q=p;*q;++q){
      if(*q=='\\'||*q=='\"')printf("\\");
      printf("%c",*q);}
    printf("%s\n",p);return 0;}
```

At least, it's 'portable' if you save it in the same encoding you will go on to run it in!

Francis: Thanks, I think that is as much as we could expect. A quine that could be compiled for one character encoding and then work with a different one seems to me to be a step too far (unless you, the reader, knows better).

The key is to put most of the program in the data string, and then print it twice, once quoted and once not quoted. Francis's example did this easily; mine uses a **for** loop to go through and deliberately quote the awkward characters.

`accu.cpp` generates the numbers **first** to **last** without **if**, **for**, **switch** or **while**. Ok, I cheated: I used the ternary operator.

Francis: Not really, but I have omitted it because this column is already about to take over this issue of *CVu*.

This does it without the ternary operator.

(This code was done in my own time and is not connected with my employer.)

```
#include <stdio.h>
#include <stdlib.h>
int f(int, int);
int g(int, int) { return 0; }
int (*which[2])(int,int) = { f, g };

int f(int first, int last) {
  printf("%d\n",first);
  int next = first - 1 + 2*(last > first);
  return (*(which[first == last]))(next,last);
}

int main(int argc, char** argv){
  f(atoi(argv[1]), atoi(argv[2]));
}
```

Francis: Great idea and results in a really simple program.

And a Java solution from Pete Disdale

Please find attached my solution – it is written in Java on the grounds that "anything not expressly forbidden is allowed". I could probably write something similar in C using linked lists but that is an exercise for a rainy day and printing the output would be a challenge.

You will need Java 7 or above to compile and run this code, although with some tweaking it should be OK with Java 5 or 6. As you can see, it uses recursion to build up the numbers to print (incrementing or decrementing) controlled by a ternary operator. The **System.out** in **main()** and the two **ret** methods are just fluff in order to use it.

No doubt it could be factored into something simpler but it appears to work as is on all the number ranges I tried.

```
import java.util.ArrayList;
import java.util.List;

public class AccuChallenge {

  public static void main(String[] args) {
    try {
      new AccuChallenge(args[0], args[1]);}
    catch (IndexOutOfBoundsException ex) {
      System.err.println("Insufficient arguments:
2 integers required.");}
  }
  private AccuChallenge(String arg1, String arg2)
  {
    int a, b;
    try {
      a = Integer.parseInt(arg1);
      b = Integer.parseInt(arg2);
```

```
        System.out.print((a <= b)
          ? new Incrementing(a, b).ret()
          : new Decrementing(a, b).ret());
      } catch (NumberFormatException nfe) {
        System.err.println("One or both arguments
are not parsable as integers.");
      }
    }
    private class Incrementing {
      private final List<Integer> result
        = new ArrayList<>();
      private final int b;
      private int a;
      public Incrementing(int a, int b) {
        this.a = a;
        this.b = b;
        addInt();
      }
      private boolean addInt() {
        return a <= b ? keepGoing()
          : printResult();
      }
      private boolean keepGoing() {
        result.add(a++);
        addInt();
        return true;
      }
      private boolean printResult() {
        System.out.println(result);
        return true;
      }
      public String ret() {
        return "";
      }
    }
    private class Decrementing {
      private final List<Integer> result
        = new ArrayList<>();
      private final int b;
      private int a;
      public Decrementing(int a, int b) {
        this.a = a;
        this.b = b;
        addInt();
      }
      private boolean addInt() {
        return a >= b ? keepGoing()
          : printResult();
      }
      private boolean keepGoing() {
        result.add(a--);
        addInt();
        return true;
      }
      private boolean printResult() {
        System.out.println(result);
        return true;
      }
      public String ret() {
        return "";
      }
    }
  }
```

Francis: I note that this is about the longest solution submitted. I wonder if that is inherent in the choice of language.

From Richard Brookfield

I managed it in C, originally by using the ternary operator, but that felt like a bit of a cheat (though you didn't prohibit it), so I refactored it out – hence the two attached versions.

Francis: I have omitted the first because of space constraints.

```
#include <stdio.h>
typedef enum tagBOOL
{
  FALSE  = 0,
  TRUE  = 1
} BOOL;
static BOOL ShowNumber(int n)
{
  printf("%d ",n);
  return TRUE;
}
static BOOL ShowRangeBase(int start, int end,
  BOOL ascending)
{
  int halfwayish  = (start+end)/2;
  return
    // Gone too far
    ascending && start>end ||
    !ascending && start<end ||
    // Something to do
    ShowNumber(start) &&
    // Maximum recursion O(log2(n))
    ascending &&
    ShowRangeBase(start+1,halfwayish,ascending) &&
    ShowRangeBase(halfwayish+1,end,ascending)||
    !ascending &&
    ShowRangeBase(start-1,halfwayish,ascending) &&
    ShowRangeBase(halfwayish-1,end,ascending);
}
static void ShowRange(int start, int end)
{
  printf("Showing range from %d to %d\n",
    start,end);
  ShowRangeBase(start,end,start<end);
  printf("\n");
}
int main(int argc, char *argv[])
{
  // Various examples and edge cases
  int i;
  for (i=-2; i<12; ++i)
  {
    ShowRange(1, i);
  }
  ShowRange(2, 1);
  ShowRange(3, 1);
  ShowRange(5, 1);

  ShowRange(-1, 2);
  ShowRange(2, -1);

  ShowRange(0, 2);
  ShowRange(2, 0);

  // Enough to show the recursion isn't deep
  ShowRange(1, 200);
  return 0;
}
```

Francis: Whilst this is a fairly long solution, it does a good deal of checking as well as testing edge cases and being adequately commented.

From Robin Williams

Francis: Though Robin has used the ternary operator, his solution is novel.

I thought of two ways to avoid control statements in C: operator `?:` and short-circuit logic. Both rely on expression evaluation, which means you need functions to return something, even if the result will be ignored (apart from generating compiler warnings).

The attached uses operator `?:` for the core logic: the short-circuit version would be more verbose. I use short-circuit logic for error-checking, in a common scripting language idiom.

It's possible to do the iteration by simple stepping rather than bisection. However, without tail-call optimization, stepping can cause the program to fail for large ranges.

```
#include <stdio.h>
#include <stdlib.h>
int rangeprint(int i1, int i2)
{
  int im = (i1+i2)/2;
  // If range has one element
  return im == i1 || im == i2 ?
  printf("%d\n",i1) :  // Print that
  // Else split
  rangeprint(i1,im) && rangeprint(im,i2);
}
int usage(char* s, int i)
{
  printf("Usage: %s <n1> <n2>\n",s);
  exit(i);
  return 1;
}
int main(int argc, char*argv[])
{
  int i1, i2;
  argc == 3 || usage(argv[0],1);
  sscanf(argv[1],"%d\n",&i1) == 1
    || usage(argv[0],1);
  sscanf(argv[2],"%d\n",&i2) == 1
    || usage(argv[0],1);
  i1 == i2 || rangeprint(i1,i2);
  printf("%d\n",i2);
}
```

Francis: Robin also sent in a system-independent quine.

```
#include <stdio.h>
int main()
{
 char s[]="#include <stdio.h>\n"
"int main()\n"
"{\n"
" char s[]=$;\n"
" for(char*t=s;*t;++t){\n"
"  if(*t=='$' && *(t+1)==';'){\n"
"   putchar('\"');\n"
"   for(char*u=s;*u;++u){\n"
"    if(*u=='\\n')printf(\"\\\\n\\\"\\n\\\"\");\n"
"    else if(*u=='\"')printf(\"\\\"\");\n"
"    else if(*u=='\\')printf(\"\\\\\");\n"
"    else putchar(*u);}\n"
"   putchar('\"');}\n"
"  else putchar(*t);\n"
" }\n"
"}\n"
"";
 for(char*t=s;*t;++t){
  if(*t=='$' && *(t+1)==';'){
   putchar('"');
   for(char*u=s;*u;++u){
    if(*u=='\n')printf("\\n\"\n\"");
    else if(*u=='"')printf("\\\"");
    else if(*u=='\\')printf("\\\\");
    else putchar(*u);}
   putchar('"');}
  else putchar(*t);
 }
}
```

A conversation piece from Steven Singer

Francis: I really enjoy this way of presenting a solution. I find them so much more fun than just cold code.

S: Let's see, you want me to write a program that takes two integer inputs (first and last), to output the integers from the first to the last and you don't want me to use if, for, switch or while.

F: Yes, and first may be higher than last.

S: There are a couple of ways of interpreting that last condition, I'll assume you mean that if `first` is higher than `last` you want me to count down. The alternative would be to count from the lower of `first` and `last` up to the higher.

F: That's a reasonable interpretation of the problem.

S: OK, let's recurse and use the ternary operator. Although the ternary operator doesn't allow us to embed statements, like `return`, we can embed function calls and that's all we need for recursion:

```
#include <stdlib.h>
#include <stdio.h>

void count(int first, int last, int step)
{
  printf("%d\n", first);
  first != last ? count(first + step, last, step)
  : (void) 0;
}
int main(int argc, char *argv[])
{
  int first = argc >= 2 ? atoi(argv[1]) : 1;
  int last = argc >= 3 ? atoi(argv[2]) : 100;
  int step = first < last ? 1 : -1;
  count(first, last, step);
  return 0;
}
```

F: What's the `(void) 0` for?

S: A ternary operator always has two value expressions but I only need one. I need to specify something in the unwanted path. The two value expressions for the ternary operator must have compatible types. Since `count()` naturally has no return value, I need the third expression also to have no return value but I can't leave it blank. Personally, I like using `(void) 0` as it makes it clear to the reader and the compiler that I really, really want to do nothing. It's very Zen.

F: Well, I suppose that works but aren't you worried about the stack usage of the recursion?

S: A little, but modern compilers are pretty good at tail call optimisations. For example, gcc 5.4.0 at optimisation level 2 produces the following on my Linux box (I'll remove some non-essential lines, like call frame information directives, alignment directives, unused labels and so on for clarity):

```
count:
  pushq %r12
  movl  %edx, %r12d
  pushq %rbp
  movl  %esi, %ebp
  pushq %rbx
  movl  %edi, %ebx
  jmp   .L3
.L6:
  addl  %r12d, %ebx
.L3:
  xorl  %eax, %eax
  movl  %ebx, %edx
  movl  $.LC0, %esi
  movl  $1, %edi
  call  __printf_chk
  cmpl  %ebp, %ebx
  jne   .L6
  popq  %rbx
```

```
  popq  %rbp
  popq  %r12
  ret
```

S: You can see the call to `printf()` but the recursive call to count has become the conditional jump to label `.L6`.

F: That looks good but what if someone feels that using a ternary operator is perhaps a little too much like an `if-else`, particularly as there are redundant expressions.

S: No problem, I can remove the ternary operators:

```
#include <stdlib.h>
#include <stdio.h>
void count(int first, int last, int step)
{
  printf("%d\n", first);
  (void) (first != last && (count(first + step,
    last, step), 0));
}
int main(int argc, char *argv[])
{
  int first = 1, last = 100, step = 1;
  (void) (argc >= 2 && (first = atoi(argv[1])));
  (void) (argc >= 3 && (last = atoi(argv[2])));
  (void) (first > last && (step = -1));
  count(first, last, step);
  return 0;
}
```

S: There you go, not a ternary operator in sight. The logical operators short-circuit paths that aren't taken so the recursion terminates properly.

F: I notice you're using a comma operator.

S: Yes, when using short circuit operators, the terms on either side can't be void expressions. Since I've declared `count()` as not having a return value, I need something to keep the compiler happy. I could have avoided the comma operator by declaring `count()` as returning `int` but that's a little unnatural as there's nothing meaningful for `count` to return.

F: OK, I see what you did there but if we're avoiding ternary operators then the short-circuit logical operators are still a bit too much like an `if`.

S: OK, fine, no short-circuiting logical or ternary operators:

```
#include <stdlib.h>
#include <stdio.h>
int opt_default(int dflt, char *argv[], int arg)
{
  return dflt;
}
int opt_parse(int dflt, char *argv[], int arg)
{
  return atoi(argv[arg]);
}
int (*opt_select[])(int, char *[], int) = {
  opt_default, opt_parse };
void nop(int first, int last, int step)
{
}
void count(int, int, int);/* Forward reference */
void (*funcs[])(int, int, int) = { nop, count };
void count(int first, int last, int step)
{
  printf("%d\n", first);
  funcs[first != last](first + step, last, step);
}
int main(int argc, char *argv[])
{
  int first = opt_select[argc >= 2](1, argv, 1);
  int last = opt_select[argc >= 3](100, argv, 2);
  int step = (first < last) * 2 - 1;
  count(first, last, step);
  return 0;
}
```

S: This time, there are no `if`-like operators at all. I've used two-element arrays of function pointers and then indexed them with boolean expressions. Element zero gets called when the expression is false and element one gets called when the expression is true.

F: Isn't that a bit obscure?

S: Actually, no. Many real systems defer complex decisions to state machines and the core of a state machine is logically a two-dimensional array of function pointers indexed by the current state and the event received. Usually there's some compression of the data structures as not all events can be received in all states. Think of the code I just gave as a state machine compressed to deal with having just one state in which it can receive events (printing a number) and two events (continue and stop).

F: I suppose that's fair.

S: Did you know that even though we're using function pointers, GCC has still performed the tail call optimisation so we won't overflow the stack:

```
count:
  pushq %r12
  pushq %rbp
  movl  %edx, %r12d
  pushq %rbx
  movl  %esi, %ebp
  movl  %edi, %ebx
  movl  %edi, %edx
  movl  $.LC3, %esi
  movl  $1, %edi
  xorl  %eax, %eax
  call  __printf_chk
  xorl  %eax, %eax
  cmpl  %ebp, %ebx
  leal  (%rbx,%r12), %edi
  setne %al
  movl  %r12d, %edx
  movl  %ebp, %esi
  popq  %rbx
  popq  %rbp
  popq  %r12
  movq  funcs(,%rax,8), %rax
  jmp   *%rax
```

S: The `jmp *%rax` is a tail call to the function pointer. You can see the three `popq`s a few lines earlier that unwind this stack frame.

F: Hmm, not all compilers are that good at tail call optimisation and it may depend on which processor is being targeted. Can you avoid recursion completely?

S: Sure, but do you mind if I go back to using ternary operators for the simple `if` conditions? I've got a choice of several options but ternary operators will make it easier to see the new code.

F: It's the least I can do.

S: OK, here goes, but be warned that since you've taken away `for`, `while` and recursion, I can think of only one way to make the loop I need:

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
  int first = argc >= 2 ? atoi(argv[1]) : 1;
  int last = argc >= 3 ? atoi(argv[2]) : 100;
  int step = first < last ? 1 : -1;
loop:
  printf("%d\n", first);
  first == last ? exit(0) : (void) 0;
  first += step;
  goto loop;
}
```

S: We couldn't use this trick for `return` as it's a statement but `exit()` is a function. C is a small language, a lot of functionality is provided by functions in libraries.

F: Exiting the entire program is a bit brute force; you can't use this trick in a function embedded in a program.

S: The rules don't say this has to be a function.

F: I suppose they don't but it'd be a better example if it were.

S: OK, how about this:

```
#include <stdlib.h>
#include <stdio.h>
#include <setjmp.h>
void count_internal(int first, int last,
  int step, jmp_buf env)
{
loop:
  printf("%d\n", first);
  first == last ? longjmp(env, 1) : (void) 0;
  first += step;
  goto loop;
}
void count(int first, int last, int step)
{
  jmp_buf env;

  setjmp(env) ? (void) 0
    : count_internal(first, last, step, env);
}
int main(int argc, char *argv[])
{
  int first = argc >= 2 ? atoi(argv[1]) : 1;
  int last = argc >= 3 ? atoi(argv[2]) : 100;
  int step = first < last ? 1 : -1;
  count(first, last, step);
  return 0;
}
```

S: There you go, no premature exit.

F: Oh good grief. That is not a better example.

S: That's the first time I've used **setjmp**/**longjmp**.

F: I'm glad to hear it.

S: It kept me entertained.

F: That's what concerns me.

S: I can do worse.

F: Oh dear.

S: How about this:

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
  int first = argc >= 2 ? atoi(argv[1]) : 1;
  int last = argc >= 3 ? atoi(argv[2]) : 100;
  int reverse = first > last;
  char buffer[256];
  sprintf(buffer, "perl -e 'print map { \"$_\\n\"
    } %s (%d..%d)'", reverse ? "reverse" : "",
    reverse ? last : first,
    reverse ? first : last);
  (void) system(buffer);
  return 0;
}
```

F: How can you justify that?

S: Like the function pointers code being an example of a more sophisticated technique (state machines) which is valid for more complex problems, sometimes it's important to realise you're using the wrong language to solve a problem. Why write a thousand lines of hard-to-debug C when it might only take a couple of lines in Perl.

F: A couple of lines of hard-to-debug Perl. You could use Python.

S: OK, a dozen lines of hard-to-debug Python. It's the programmer that makes code hard-to-debug, not the language.

F: Let's not start a flame war.

S: Spoilsport. But even if you don't go to another language, you should look at the libraries for the language you're using. In the Perl example, the loops and the iteration are implicit. Many languages have similar higher-level constructs. C has a smaller set, but there are some useful routines there (not to mention a wide variety of third party libraries). For example, if you wanted to sort an array in C you wouldn't write your own sort routine without having a story as to why the library **qsort()** routine wasn't suitable. Writing sort routines from scratch is error prone; it's better to avoid it if possible. Less well-known are functions for searching flat arrays either linearly or using binary search. There are even functions for managing binary trees and hash tables. For example, we can solve the problem with this program, which you could imagine as a learning exercise to investigate the order in which **lfind()** visits a linear array just with the output slightly manipulated to give the numbers we want:

```
#include <stdlib.h>
#include <stdio.h>
#include <search.h>
struct params {
  int first;
  int step;
  char *base;
};
int compar(const void *key, const void *velem)
{
  const struct params *pp = key;
  const char *elem = velem;
  int index = (int) (elem - pp->base);
  printf("%d\n", index * pp->step + pp->first);
  return 1; /* Keep going */
}
int main(int argc, char *argv[])
{
  struct params p;
  int last;
  size_t num;
  p.first = argc >= 2 ? atoi(argv[1]) : 1;
  last = argc >= 3 ? atoi(argv[2]) : 100;
  p.step = p.first < last ? 1 : -1;
  num = (last - p.first) * p.step + 1;
  p.base = malloc(num);
  (void) lfind(&p, p.base, &num, 1, compar);
  return 0;
}
```

F: I notice you didn't initialise the array you allocate.

S: There's no need as I never actually dereference the values. I just print out the order in which the array indexes are visited.

F: I guess that's technically correct, even if it's a little ugly. You are using a lot of memory for no benefit.

S: I wanted to keep it legal so I needed to ensure the pointers **p.base** and **elem** pointed to the same object otherwise the subtraction wouldn't be defined in the C specification. Some equivalent techniques in other languages might suffer the same problem; for example, building the entire output in memory before writing it out consumes memory. Sometimes this is a problem and using iterator functions or objects can help. Other times, the problem you're solving requires having all the data in memory so you have to pay that cost anyway. However, overall, I agree for this problem it's ugly and overkill. I don't even test whether the allocation succeeded.

F: It could be worse, at least you didn't abuse the C pre-processor.

S: Did someone say "abuse the C pre-processor":

```
#include <stdlib.h>
#include <stdio.h>
#define PASTE(a, b) b ## a
```

```
#define until(x) PASTE(ile, wh) (!(x))
int main(int argc, char *argv[])
{
  int first = argc >= 2 ? atoi(argv[1]) : 1;
  int last = argc >= 3 ? atoi(argv[2]) : 100;
  int step = first < last ? 1 : -1;
  do {
    printf("%d\n", first);
    first += step;
  } until(first == last + step);
  return 0;
}
```

F: I should have kept my mouth shut.

S: Hey, it obeys the rules, it doesn't contain the word 'while', it doesn't even include the letters w, h, i, l and e in that order.

F: Can we stop now?

S: Nope. A coding competition isn't over until someone abuses the rules.

F: And that last example didn't already abuse the rules?

S: Let's pretend it's an important real-world lesson about requirements. What did the rules say the program had to output?

F: "the integers from the first to the last".

S: Thanks for the quotation marks:

```
#include <stdio.h>
int main(void)
{
  printf("the integers from the first to the
last\n");
  return 0;
}
```

S: Does it help if I pretend it's an important lesson about escaping strings to avoid SQL injection attacks and similar.

F: Not really. Please can we stop now?

S: Alright. I've had my fun.

F: Yes, you seemed to be enjoying this all too much.

From Stephen Baynes

Francis: Here is an interesting collection of (short) solutions.

The problem is not how to do it at all, C provides short circuit operators that combined with recursion easily do the job (1, 2, 3) but how many other ways it can be done. One can terminate a **goto** loop by terminating the program with assert (4), division by zero [undefined behaviour so will depend on implementation] (5) or by the process killing itself using some clever arithmetic using booleans as integers to calculate signal number that does or does not stop execution [requires POSIX] (6). Using booleans as integers can be used to select a function to call from a table that recurses or not to end the loop (7). It should be possible to recursively use the error callback of **memset_s** to provide a controlled recursive loop (8) but I was unable to find a compiler that implements this optional feature so this code is untested.

One could do all sorts of non-portable implementations using system but that is outside the spirit of doing it in C.

I wondered about printing consecutive numbers from the comparison function passed to **qsort**. **qsort** on an array of size *n* would give you at least *n*-1 comparisons but could give more though a finite amount. The printing would terminate, but it would be very difficult to ensure it terminated at the right time and when it terminated would not be portable.

Having to read in the **first** and **last** values means one cannot use recursive pre-processor macros to solve the challenge.

Francis: Which is just as well because this column is already heading for record-breaking size.

1)

```
#include <stdio.h>
static int from, to;
int pri( int n ){
  printf( "%d\n", n );
  n < to && pri( n + 1 );
  return n;
}
int main(){
  scanf( "%d %d", &from, &to );
  pri( from );
  return 0;
}
```

2)

```
#include <stdio.h>
static int from, to;
int pri( int n ){
  printf( "%d\n", n );
  n >= to || pri( n + 1 );
  return n;
}
int main(){
  scanf( "%d %d", &from, &to );
  pri( from );
  return 0;
}
```

3)

```
#include <stdio.h>
static int from, to;
int pri( int n ){
  printf( "%d\n", n );
  n < to ? pri( n + 1 ) : 0;
  return n;
}
int main(){
  scanf( "%d %d", &from, &to );
  pri( from );
  return 0;
}
```

4)

```
#include <stdio.h>
#include <assert.h>
static int from, to;
int main(){
  scanf( "%d %d", &from, &to );
  int n = from;
LOOP:
  printf( "%d\n", n );
  assert( n < to );
  ++n;
  goto LOOP;
  return 0;
}
```

5)

```
#include <stdio.h>
static int from, to;
static int j = 0;
int main(){
  scanf( "%d %d", &from, &to );
  int n = from;
LOOP:
  printf( "%d\n", n );
  j =  1 / ( to -n );
// This might depend on your compiler
  ++n;
  goto LOOP;
  return 0;
}
```

6)

```
#include <sys/types.h>
```

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
static int from, to;
int main(){
  scanf( "%d %d", &from, &to );
  int n = from;
  pid_t pid = getpid();
LOOP:
  printf( "%d\n", n );
  int j = (int)( n >= to );
  kill( pid, j * SIGTERM ) ;
    // Signal 0 does not signal.
  ++n;
  goto LOOP;
  return 0;
}
```

7)

```
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
static int from, to;
typedef void action( int );
action *jump_table[2];
static void action_end( int n ) { }
static void action_next( int n ) {
  printf( "%d\n", n );
  int j = (int)( n >= to );
  jump_table[ j ]( n + 1 );
}
int main(){
  scanf( "%d %d", &from, &to );

  jump_table[ 0 ] = action_next;
  jump_table[ 1 ] = action_end;
  action_next( from );
  return 0;
}
```

8)

```
// This code is not tested as my compiler
// does not support memset_s
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
static int from, to;
static int n;
void pri( void ){
  printf( "%d\n", 1 + from - n );
  --n;
  char c;
  memset_s( &c, 1, 0, n );
}
void ch(
  const char *restrict msg,
  void *restrict ptr,
  errno_t error
){
  pri()
}
int main(){
  scanf( "%d %d", &from, &to );
  n = 1 +  from - to;
  set_constraint_handler_s( ch )
  pri( );
  return 0;
}
```

From Tim Kent

My first thought was how to control program flow without loops and **if** statements and it didn't take a moment to recall a code base that I inherited which uses exceptions for normal code flow. I ignored the use of the ternary (**?:**) operator thinking that this was not in the spirit of the challenge. So how could I use exceptions to control a loop? The first thing that came to mind was divide by zero. This led me to the following solution:

```
#include <iostream>
int main()
{
  /*Here is an initial idea using divide by zero,
    to solve Challenge 4 in CVU Vol 30 Issue 3,
    July 2018 */
  int first = 1;
  int last = 100;
  /*check that first is less than last otherwise
    fault on divide by zero to terminate program.
    TRUE - TRUE = 0*/
  volatile int terminating_calc1 =
    1/ ((first > last) - true);
loop:
  std::cout << first++ << '\n';
  volatile int terminating_calc2 =
    1/(last-first+1);
    /* fault terminate with divide by zero after
       we reached the last number */
  goto loop;
  return 0;
}
```

The only problem is that this signals and terminates [well, you hope it does that rather than rewrite your hard-drive ☺] the program meaning this could not be used in a larger program. Sure, it would be possible to handle the signal and not terminate but that would be platform dependent code. How to conditionally throw an exception? I almost have this condition in the program above. **True - True == 0**, **True - False == ?**. If I could force **True - False == 1** then I could use a two-element array of functions to throw or not throw based on the index. I ended up just dividing by true to achieve this. The only other question in my mind was what to throw. I arbitrarily chose a **void** pointer so as not to interfere with any other code that might be used with this solution (not that I am recommending using this solution anywhere!), which leads to the following solution:

```
void throw_if_true(bool condition)
{
  typedef void fntype();
  static fntype* fn_lut[2] = { []{
    throw static_cast<void*>(0);}, []{}};
  int index = (true - condition)/true;
  fn_lut[index]();
}

int main()
{
/*Using an exception for flow control, to solve
  Challenge 4 in CVU Vol 30 Issue 3, July 2018 */
  int first = 1;
  int last = 100;
  try{
    throw_if_true(first > last);
    loop:
    std::cout << first++ << '\n';
    throw_if_true(first > last);
    goto loop;
  }catch(...) {}
  return 0;
}
```

Now that this works, can I go a step further and avoid signals and exceptions altogether? Building on what I have so far, I thought about replacing the lambda that throws with a lambda that recurses, and this led eventually to the following solution.

```
#include <iostream>
typedef void fntype(int,int);
void count_inner(const int first,
  const int last);
static fntype* fn_lut[2] = {
  [](const int first, const int last){},
  [](const int first, const int last)
  {count_inner(first+1, last);} };
void count_inner(const int first, const int last)
{
  int index = (true - (first > last))/true;
  std::cout << first << '\n';
  fn_lut[index](first, last);
}
void count(const int first, const int last)
{
  int index = (true - (first > last))/true;
  fn_lut[index](first-1, last-1);
}
int main()
{
// Using recursion  for flow control, to solve
// Challenge 4 in CVU Vol 30 Issue 3, July 2018
  int first = 1;
  int last = 100;
  count(first, last);
  return 0;
}
```

From Colin Hersom

In *CVu* 30.3, you challenged us to devise a program to output a sequence of numbers without using **for**, **if**, **switch** and **while**. You also made an implicit challenge to do it in C (by stating that you could not) as well as C++. Now, I know that you are a very clever man and know the intricacies of C in far more detail than most, so I was surprised at the statement that you could not do it in C. If this was meant as a provocation, it worked!

Francis: Hmm… I am getting old and forgetful. For example I forgot to exclude the ternary operator. ☹

I agree that recursion (or indeed anything) cannot know when to stop without a condition (unless you cause the program to crash!), but you have not excluded conditions, only those keywords. There is still the conditional operator, and I have used this in my solution [omitted to save space]. The problem with this is that the recursion will take up a lot of stack space and if you ask for a very large range, the system may run out of space and cause an error. With my system, it crashes when asked to produce a sequence of around 261,850 numbers, but varies from run to run. [You need a better compiler that spots tail recursion. ☺]

So to C++. My first solution was to create a vector with all the numbers in and then output the vector. This is OK, but also uses stack space, although not so much. However, the vector is being created and then written, without any other use, so it seemed sensible to remove the vector completely and replace it with an iterator that produced the correct sequence, outputting that directly.

Francis: Both of Colin's C++ solutions use the ternary operator in the set-up in **main** but the actual method for creating the desired output do not. I have provided them because they have interesting mechanisms.

```
class sequence_interator :
  public   iterator<forward_iterator_tag, int>
{
public:
  sequence_interator(int start, int end)
    : current(start)
    , last(end)
    , inc(start < end ? 1 : -1)
  {  }
  sequence_interator &begin()
  {
    return *this;
  }
  sequence_interator &end()
  {
    return *this;
  }
  // The compared interator is always 'this' so
  // no need to examine it
  // Just using to test for the last one
  bool operator !=(const sequence_interator&)
  {
    return last != current - inc;
  }
  sequence_interator &operator++()
  {
    current += inc;
    return *this;
  }
  int operator *() const
  {
    return current;
  }
private:
  int current;
  int last;
  int inc;
};
int main(int av, char **ac)
{
  int start = av >1 ? atoi(ac[1]) : 0;
  int end = av >2 ? atoi(ac[2]) : 100;
  int inc = start < end ? 1 : -1;
  sequence_interator seq(start, end);
  copy(seq.begin(), seq.end(),
    ostream_iterator<int>(cout, " "));
  cout << endl;
  return 0;
}
```

From Burkhard Kloss

Spurred on by Francis's challenge, I dusted off the C compiler and wrote my first C program in ... over a decade at least? Turns out I can still do it, although I wouldn't say I'm particularly proud of this one. It does work, though, and matches the spec as far as I can tell. For the simple case, this should do the trick.

```
#include <stdio.h>
#include <stdlib.h>
int quitter()
{
  exit(0);
  return -1;
}
int main()
{
  int i = 1;
start:
  printf("%d\n", i);
  i = i < 100 ? i + 1 : quitter();
  goto start;
}
```

Francis: Yes, but I forgot to exclude the ternary operator. Your solution can be amended to avoid use of the ternary operator by replacing that statement with:

```
i < 100 || quitter(); ++i;
```

I notice that the ternary operator can almost invariably be replaced by lazy evaluation. For example, usually this:

```
expr1 ? expr2 : expr3
```

can be transformed into:

```
(expr1 && expr2) ||| expr3;
```

Using the same approach for the extended problem with inputs of `first` and `last` on the command line – even including rudimentary input checking:

```
#include <stdio.h>
#include <stdlib.h>
int quitter()
{
  exit(0);
  return -1;
}
int sign(int x)
{
  return x > 0 ? 1 : -1;
}
int noisy_quit(char const * msg)
{
  puts(msg);
  exit(1);
  return -1;
}
void check_args(int argc)
{
  argc == 3 ? 0 : noisy_quit
    ("first_to_last start stop");
}
int main(int argc, char * argv[])
{
  check_args(argc);
  int start = atoi(argv[1]);
  int stop = atoi(argv[2]);
  int step = sign(stop - start);
  int i = start;
loop:
  printf("%d\n", i);
  i = i != stop ? i + step : quitter();
  goto loop;
}
```

Not the world's most elegant code, but it seems to handle all possible conditions, and has no sign of `if`, `for`, `while`, or `switch`.

And the winner is...

Well, none of the above. I have asked Steve to turn the winning submission (from Andreas Gieriet) into a free-standing article, which you will find on page 6.

Thanks to all of you who have put in so much thought and ingenuity. It is eye-opening to see how many completely different solutions you collectively managed to find.

If you read something in C Vu that you particularly enjoyed, you disagreed with or that has just made you think, why not put pen to paper (or finger to keyboard) and tell us about it?

Challenge 5

Many years ago someone – I have forgotten who – ran a competition for writing a (natural) language that only had seven words. The trap was to assume that each word had to have a single meaning. To deal with such a paucity of words requires making good use of combinations. Start with two words meaning one and zero then using binary you have an unlimited supply of numbers. Now make a third word mean 'select meaning *n* from the next word' and you have an unlimited supply of meanings available to you (and only using four words … and yes, I know that I could do the same with only three words).

However, I am going to test your ingenuity by asking you to select seven operators with their normal meanings. No grotesque multi-purpose operators. So if `+` is one of your operators, `a+b` will mean add 'a to b')

Having chosen your seven operators, show how you would obtain as many other operators from C++ as you can. To get you started: Suppose that one of your seven operators is `-` then you could write:

```
template<typename T> T operator+(T t1, T t2)
  { return t1 - (0 - t2)};
```

I am pretty sure you cannot find a subset of seven operators that spans the set of C++ operators but I wonder how far you can get.

Submissions to francis.glassborow@gmail.com

Bookcase

The latest roundup of book reviews.

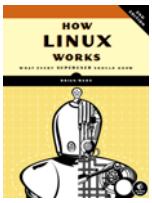
If you want to review a book, your first port of call should be the members section of the ACCU website, which contains a list of all of the books currently available. If there is something that you want to review, but can't find on there, just ask. It is possible that we can get hold of it.

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

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

Astrid Byro (astrid.byro@gmail.com)

How Linux Works



By Brian Ward. ISBN 978-1-59327-567-9

Reviewed by Ian Bruntlett

This book describes itself as 'What Every Superuser Should Know'. I read it very studiously, making notes and occasionally branching out to other books (in particular *Networking for Systems Administrators* by Michael W. Lucas) and reading relevant man and info pages. It acts as a framework for my Linux knowledge, with a view to adding to it. ☺

Structurally, it has 17 chapters, a Bibliography and a reasonable index.

Chapter 1, 'The Big Picture', gives an overview of things. Useful info for people who don't have low-level knowledge of their computers or who gained their low-level knowledge on other systems.

Chapter 2, 'Basic Commands and Directory Hierarchy', covers the use of fundamental file-handling and other frequently used commands, shell globbing and, very helpfully goes into Unix matters like dot files, environment vs shell variables, special characters, I/O redirection, process manipulation, file modes and permissions, symbolic links, archive files, an overview of the Linux filesystem hierarchy and the use of **`sudo`** to run commands with superuser privileges.

Chapter 3, 'Devices', explains how the 'everything is a file' idea is useful in Linux. It covers the basics (which is all most people will ever need to know) and goes into detail which is interesting. I particularly enjoyed the explanation of the Linux SCSI subsystem and how it is co-opted to handle USB devices as well. Chapter 4, 'Disks and Filesystems', builds upon Chapter 3, going into much more technical information which might be of interest to someone wanting to do kernel things.

Chapter 5, 'How the Linux Kernel Boots' discusses how a PC typically starts up from the BIOS running until user space (i.e. **`init`**) starts. Boot loaders are discussed but the lion's share of this covers GRUB – from a gentle introduction to writing your own menu entries to installing GRUB by hand. The two different boot loaders (BIOS MBR, UEFI) are covered.

Chapter 6, 'How User Space Starts', covers **`init`**, runlevels, has 16 dense pages about **`systemd`** (which appears to be the successor to the old init). It also covers the old **`init`** and, **`upstart`** (which I believe is no longer used). At the end it even covers the Initial RAM Filesystem.

Chapter 7, 'System Configuration: Logging, System Time, Batch Jobs and Users'. Discusses the System Logger, how your hard drive doesn't get filled to overflowing with log files, the role of many of the strange configuration files in `/etc` – in particular `/etc/passwd`, `/etc/group`, how the login sequence works, Network Time. The cron system for scheduling regular tasks using **`cron`**, **`crontab`** and **`at`**. The various user ID types are discussed (Effective UID, Real UID and Saved UID). PAM – Pluggable Authentication Modules is also discussed.

Chapter 8, 'A closer look at processes and resource utilization' is a useful chapter – educational and of practical use when trying to track down performance problems. A variety of commands (**`lsof`**, **`strace`**, **`ltrace`**, **`ps`**, **`top`**, **`uptime`**, **`vmstat`**, **`iostat`**, **`pidstat`**) are explained.

Chapter 9, 'Understanding your network and its configuration' is a bold title for a bold chapter and chapters 10 and 12 build further on these foundations It asks two questions – 'How does the computer sending the data know where to send its data?' and 'When the destination receives the data, how does it know what it just received?' All the basics of networking (Packets, Network Layers, IP, subnets, routing, ping, traceroute, kernel network interfaces, NetworkManager, DNS, config files, TCP, UDP, DHCP, Linux as a Router, IP Masquerading, Firewalls, Ethernet, ARP WiFi) are covered.

Chapter 10, 'Network Application and Services' mainly covers TCP services and illustrates how to use telnet to interact with a webserver. The use of Secure Shell (SSH) and many of its uses. Diagnostic tools (**`netstat`**, **`lsof`**, **`tcpdump`**, **`netcat`** and **`nmap`**) are covered. RPC is covered, briefly. There is a good section on Network security. The role of TCP socket and Unix Domain sockets in handling applications on the localhost or other hosts is explained (liked it).

Chapter 11, 'Introduction to shell scripts' is the weakest chapter. It discusses the use of the Bourne Shell (sh) rather than the Bourne Again Shell (bash) which is a rather curious choice for a Linux command line interpreter. Despite that, it is a reasonable introduction to shell scripting, with the notable exception that it doesn't discuss debugging shell scripts. It does encourage the reader to use a scripting language like Python once a shell script has become too complicated and slow.

Chapter 12, 'Moving files across the network' builds on previous chapters, covering file and printer sharing (SimpleHTTPServer, rsync, SMB, NFS).

Chapter 13, 'User Environments'. No, not GNOME – dot files and bash configuration files.

Chapter 14, 'A Brief Survey of the Linux Desktop' discusses different components. There is the desktop (Window Managers, Toolkits, Desktop Environments, Applications), a look at X and some related commands, the D-Bus message passing system and printing (CUPS).

Chapter 15, 'Development Tools' – quite important. Does a decent description of compiling C programs, covers static and shared libraries, **`make`**, debuggers, and mentions other scripting languages.

Chapter 16, 'Introduction to compiling software from C source code' does a reasonable job of explaining how to compile software distributed as open source. It covers **`make`**, GNU Autoconf, pkg-config, patch.

Chapter 17, 'Building on the Basics' is a 'where next?' chapter, asking (and answering) what to look at next.

I suspect that the 3rd edition of the book, if forthcoming, would cover systemd in more depth and use bash instead of sh..

Overall, I was thrilled with this book. It has consolidated some of my Linux knowledge and expanded other parts of it. Well worth reading.

View from the Chair

Bob Schmidt
chair@accu.org

YouTube

ACCU has a fair amount of content on YouTube, consisting of the recordings of ACCU conference sessions from this year [1], and going back to 2016. It is a tremendous resource for those who can't make it to the conference, and for those who can't be in five places at once during the conference. This year the videos were produced by Jim Roper. Please join me in thanking Jim for the fine work he has done on these videos.

While on the subject of the videos, it seems that some viewers are using the comments sections of the videos to ask questions. You should be aware that ACCU does not monitor the comments of any of the videos – it is up to the speakers to do so if they wish. In the future we may disable comments in order to not mislead viewers into believing they will always get an answer.

GDPR

ACCU's Privacy Policy [2], our response to the EU's GDPR, has continued to be refined over the past two months. The current version has been approved by the committee, and has been moved from draft to final status. While we don't foresee any major changes to the policy, we may refine the document as we all gain experience with the GDPR and its mandates.

Once again, our thanks go out to Nigel Lester and everyone who assisted him in the drafting of our Privacy Policy.

ePub

We are slowly getting caught up on the production of ePub versions of ACCU's magazines, thanks to the efforts of Daniel James. I didn't prioritize generating the ePub versions when I started acting as web editor, since they had been labelled 'experimental'. Through our conversations and correspondence, Daniel has educated me that ePubs are a favoured format for some of our membership, and I'm pleased he has picked up this portion of the web editing function.

We are slowing introducing changes to the way magazine content is presented through the website. Currently the 'by cover' index takes a user directly to the PDF version, or to the ePub version if there are two copies of the magazine cover. In the future the 'by cover' magazine indexes that will take the user to a page that will present all three versions of a magazine's content – PDF, ePub, and HTML. Hopefully this will make it easier for you to find your preferred presentation type.

IncludeCPP

IncludeCPP is a "global, inclusive, and diverse community for developers interested in C++" [3]. Their goal is to help make the C++ community more inclusive. To that end, their website includes resources for employers and conference organizers. Their mission aligns nicely with ACCU's core values, as expressed in our Diversity Statement [4].

If you look at the picture at the top of IncludeCPP's Twitter page [5], you will recognize among the group several people who are members of ACCU and/or frequent contributors to our magazines and conferences. Please consider adding your voice to your colleagues' in supporting diversity in the industry.

Finances

ACCU's 2017 financial statement was finalized in May, and shows that our financial situation remains strong. Details can be found in the 2018 AGM Pack [6] (they are labelled 'draft', but the numbers didn't change in the final version). Here are the major numbers for 2017, in British Pounds:

Revenue	40,902.00
Cost of Sales	(30,839.00)
Expenses	(1,191.00)
Interest	2.00
Surplus	8,874.00

As in the past several years, the cost of producing *CVu* and *Overload* is our major expense, and ACCU's share of the income from the 2017 conference represents a large percentage of our operating surplus.

Call for volunteers

We still need volunteers. Are you up for the challenge? E-mail me at chair@accu.org.

- Web Editor
- Book Reviews
- Social Media
- Publicity
- Study Groups

References

[1] ACCU 2018 – https://www.youtube.com/channel/UCJhay24LTpO1s4bIZxuIqKw
[2] ACCU Privacy Policy – https://accu.org/index.php/privacy_policy
[3] IncludeCPP – http://www.includecpp.org/
[4] ACCU Diversity Statement – https://accu.org/index.php/aboutus/diversity_statement
[5] #include_cpp – https://twitter.com/include_cpp
[6] AGM Pack (members only) – https://accu.org/content/agm/AGM-2018-Pack.pdf

Members' News

Fran Buontempo's book, *Genetic Algorithms and Machine Learning for Programmers*, aims to help the reader discover machine-learning algorithms using a handful of self-contained recipes to code your way out of a paper bag.

It is available in beta here: https://pragprog.com/book/fbmach/genetic-algorithms-and-machine-learning-for-programmers.

You can download three small samples through the link or buy a beta eBook. The final paper book should be available before Christmas.