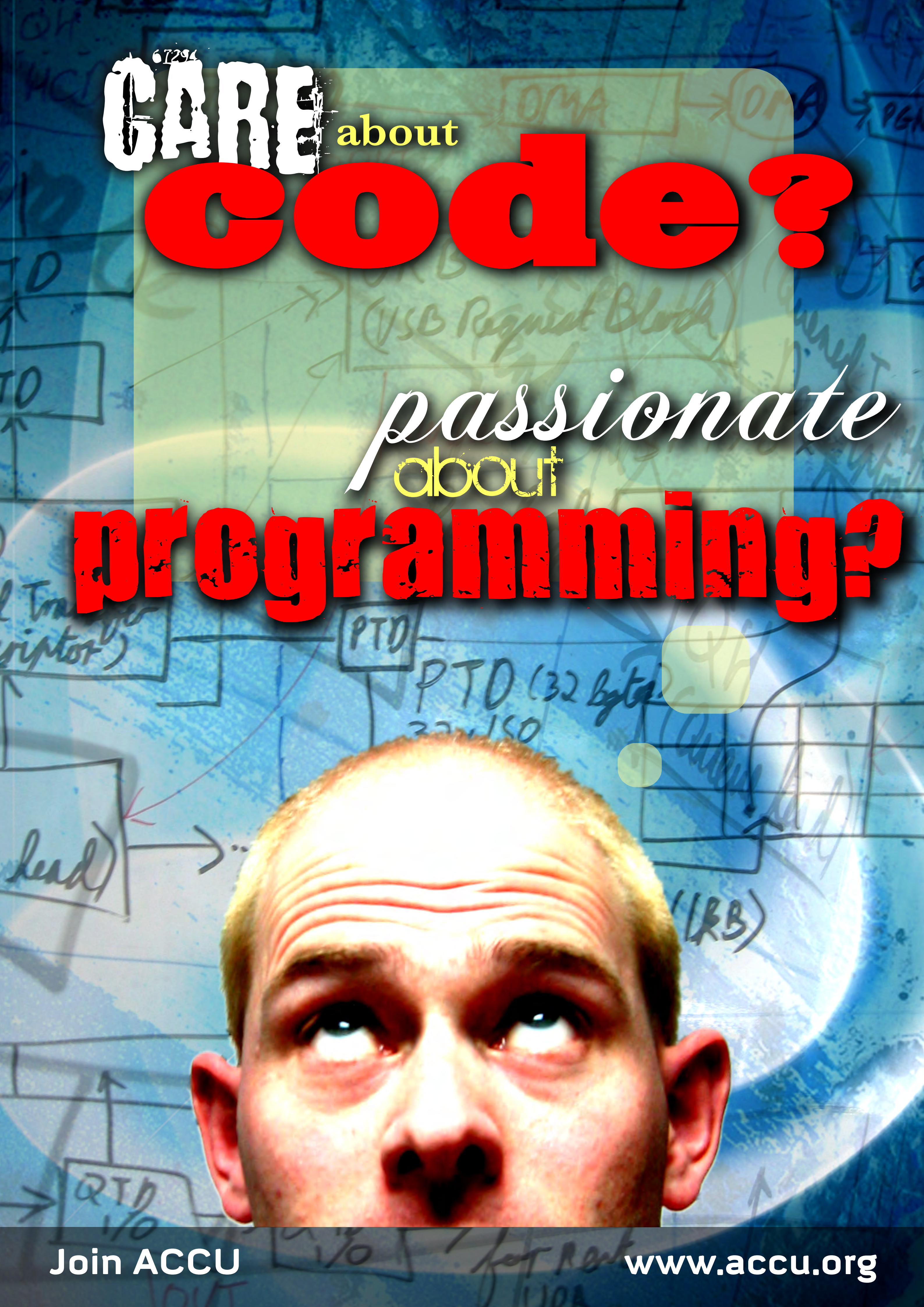
the magazine of the accu

www.accu.org

{cvu}

Volume 29 • Issue 5 • November 2017 • £3


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

{cvu}

Volume 29 Issue 5
November 2017
ISSN 1354-3164
www.accu.org

Editor
Steve Love
cvu@accu.org

Contributors
Andy Balaam, Silas S. Brown, R. Brian Clark, Francis Glassborow, Pete Goodliffe, Roger Orr, A Student

ACCU Chair
Bob Schmidt
chair@accu.org

ACCU Secretary
Malcolm Noyes
secretary@accu.org

ACCU Membership
Matthew Jones
accumembership@accu.org

ACCU Treasurer
R G Pauer
treasurer@accu.org

Advertising
Seb Rose
ads@accu.org

Cover Art
Pete Goodliffe

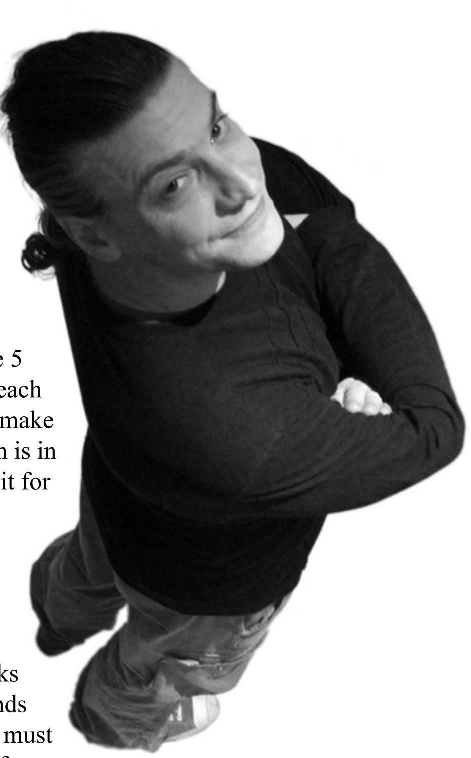
Print and Distribution
Parchment (Oxford) Ltd

Design
Pete Goodliffe

accu

Fiction

How long do you think it would take to drive from London, capital city of the UK, to Florence, centre of culture and fine art in Italy? Piece of string calculation with an Atlas (old school!) says it's around 750 miles. Average speed on the good roads of France and Italy should be about 50mph, so that's 15 hours. Add a couple for good measure, we could still do it in one day, right? We both know how to drive a car, so we can share the driving.



The GPS reports that we've underestimated the distance by well over 200 miles. That'll add maybe 5 hours to the journey, but we can manage 11 hours each in stints of 5 hours or so, and we should be able to make up some time on the motorways. The first real hitch is in leaving the UK – without a booking we need to wait for a free slot on a boat. 2 hours wasted, and most of 2 hours on a boat – that's almost 20% of our time budget on about 10% of the distance!

France is a breeze and we swap places near the Swiss border. We've made up some time, but it's getting dark. The Swiss border guard politely checks our driving licences, and yours has expired. He hands them back, and just as politely makes it clear that I must do the driving. After another hour we have to stop for strong coffee. And a quick nap.

It is starting to become clear to me that my initial estimate of 1 day is looking increasingly ambitious, and there are still so many things that can happen: breakdown, accident, diversion – we're not over the Alps yet – and with another 6 hours (at least!) of driving, I'm going to need a good rest before attempting it.

Which means we'll be getting to Milan in the rush-hour...

Knowing how to drive a car might be a little bit like knowing how to write code in a given programming language, and marking way-points on a map (or GPS) like sketching a project plan. Predicting future events is a whole different thing. And as for predicting *when* they will happen, that's just for fiction.

Steve

STEVE LOVE
FEATURES EDITOR

The official magazine of ACCU

ACCU is an organisation of programmers who care about professionalism in programming. That is, we care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

ACCU exists for programmers at all levels of experience, from students and trainees to experienced developers. As well as publishing magazines, we run a respected annual developers' conference, and provide targeted mentored developer projects.

The articles in this magazine have all been written by programmers, for programmers – and have been contributed free of charge.

To find out more about ACCU's activities, or to join the organisation and subscribe to this magazine, go to www.accu.org.

Membership costs are very low as this is a non-profit organisation.

DIALOGUE

REGULARS

FEATURES

SUBMISSION DATES

C Vu 29.6:	1st December 2017
C Vu 30.1:	1st February 2017
Overload 142:	1st January 2017
Overload 143:	1st March 2018

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

Code Aesthetics

Pete Goodliffe implores us to care (enough) about code beauty.

Appearances are deceptive.
Aesop

Code aesthetics are the most immediate determinant of how easy a section of code will be to work with. No one likes working with messy code. No one wants to wallow in a mire of jagged, inconsistent formatting, or battle with gibberish names. It's not fun. It's not productive. It's the programmer's purgatory.

Sadly, programmers care so much about code presentation that they end up bickering about it. This is the stuff that holy wars are made of. That, and which editor is best [1]. Tabs *versus* spaces. Brace positioning. Columns per line. Capitalisation. I've got my preferences. You have yours.

Godwin's law states that as any discussion on the Internet grows longer, the probability of a comparison to the Nazis or Hitler approaches one. *Goodliffe's law* (unveiled here) states that as any discussion about code layout grows, the probability of it descending into a fruitless argument approaches one.

Good programmers care deeply about good code presentation. But they rise above this kind of petty squabble. Let's act like grown-ups.

Stop fighting over code layout. Adopt a healthy attitude to your code presentation.

Our myopic focus on layout is most clearly illustrated by the classic dysfunctional code review. When given a section of code, the tendency is to pick myriad holes in the presentation. (Especially if you only give it a cursory skim-read, then layout is all you'll pick up on.) You feel like you've made many useful comments. The design flaws will be completely overlooked because the position of a bracket is wrong. Indeed, it seems that the larger the code review, and the faster it's done, the more likely this blindness will strike.

Presentation is powerful

We can't pretend that code formatting is unimportant. But understand why it matters. A good code format is *not* the one you think looks prettiest. We do not lay out code in order to exercise our deep artistic leanings. (Can you hear the code-art critics? Daaaahling, look at the wonderful Pre-Raphaelite framing on that nested switch statement. Or: you have to appreciate the poignant subtext of this method. I think not.)

Good code is clear. It is consistent. The layout is almost invisible. Good presentation does not draw attention or distract; it serves only to reveal the code's intent. This helps programmers work with the code effectively. It reduces the effort required to maintain the code.

Good code presentation reveals your code's intent. It is not an artistic endeavour.

Good presentation techniques are important, not for beauty's sake, but to *avoid mistakes* in your code. As an example, consider the following C snippet:

```
bool ok = thisCouldGoWrong();
if (!ok)
    fprintf(stderr, "Error: exiting...\n");
    exit(0);
```

You can see what the author intended here: `++exit(0)++` was only to be called when the test failed. But the presentation has hidden the real behaviour: the code will always `++exit++`. The layout choices have made the code a liability. [2]

Names have a similarly profound effect. Bad naming can be more than just distracting, it can be downright dangerous. Which of these is the bad name?

```
bool numberOfGreenWidgets;
string name;
void turnGreen();
```

The `numberOfGreenWidgets` is a variable, right? Clearly a counter is not represented by a boolean type. No; it's a trick question. They're all bad. The string does not actually hold a name, but the name of a colour; it is set by the `turnGreen()` function. So that variable name is misleading. And `turnGreen` was implemented thus:

```
void turnGreen()
{
    name = "yellow";
}
```

The names are all lies!

Is this a contrived example? Perhaps; but after a little careless maintenance, code can quickly end up in this state. What happens when you work with code like this? Bugs. Many, many bugs.

We need good presentation to avoid making code errors. Not so we can create pretty ASCII art.

Encountering inconsistent layout and hodgepodge naming is a sure sign that code quality is not high. If the authors haven't looked after the layout, then they've probably taken no care over other vital quality issues (like good design, thorough testing, etc.).

It's about communication

We write code for two audiences. First: for the compiler (or the language runtime). This beast is perfectly happy to read any old code slop and will turn it into an executable program the only way it knows how. It will impassionately do this without passing judgment on the quality of what you've fed it, nor on the style it was presented in. This is more a conversion exercise than any kind of code 'reading'.

The other, more important, audience is *other programmers*. We write code to be executed by a computer, but to be *read* by humans. This means:

- You right now, as you're writing it. The code has to be crystal clear so you don't make implementation mistakes.
- You, a few weeks (or months) later as you prepare the software for release.
- The other people on your team who have to integrate their work with this code.
- The maintenance programmer (which could be you or another programmer) years later, when investigating a bug in an old release.

Code that is hard to read is hard to work with. This is why we strive for clear, sympathetic, supporting presentation.

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

Remember who you're writing code for: other people.

We've already seen that code can look pretty but obscure its intent. It can also look pretty, but be unreasonably hard to maintain. A great example of this is the 'comment box'. Some programmers like to present banner comments in pretty ASCII-art boxes:

```
 /***********************************************
   * This is a pretty comment.                  *
   * Note that there are asterisks on the right-  *
   * hand side of the box. Wow; it looks neat.    *
   * Hope I never have to fix this tiypo.         *
************************************************/
```

It's cute, but it's not easy to maintain. If you want to change the comment text, you'll have to manually rework the right-hand line of comment markers. Frankly, this is a sadistic presentation style, and the people who choose it do not value the time and sanity of their colleagues. (Or they hope to make it so crushingly tedious to edit their comments that no one dare adjust their prose.)

Layout

Code layout concerns include indentation, use of whitespace around operators, capitalisation, brace placement (be it K&R style, Allman, Whitesmith, or the like), and the age-old tabs *versus* spaces indent debate. In each of these areas there are a number of layout decisions you can make, and each choice has good reasons to commend it. As long as your layout choices enhance the structure of your code and help to reveal the intent, then they're good.

A quick glance at your code should reveal the shape and structure. Rather than argue about brace positioning, there are more important layout considerations, which we'll explore in the following sections.

Structure well

Write your code like you write prose.

Break it up into chapters, paragraphs, and sentences.
Bind the like things together; separate the different things. Functions are akin to chapters. Within each chapter may be a few distinct but related parts of code. Break them up into paragraphs by inserting blank lines between them. Do not insert blank lines unless there is a natural 'paragraph' break. This technique helps to emphasise flow and structure.

For example:

```
void exampleFunction(int param)
{
  // We group things related to input
  param = sanitiseParamValue(param);
  doSomethingWithParam(param);

  // In a separate "paragraph" comes other work
  updateInternalInvariants();
  notifyOthersOfChange();
}
```

The order of code revelation is important. Consider the reader: put the most important information first, not last. Ensure APIs read in a sensible order. Put what a reader cares about at the top of your class definition. That is, all public information comes before private information. Creation of an object comes before use of an object.

This grouping might be expressed in a class declaration like the one in Listing 1.

Prefer to write shorter code blocks. Don't write one function with five 'paragraphs'. Consider splitting this up into five functions, each with a well-chosen name.

Consistency

Avoid being precious about layout styles. Pick one. Use it consistently. It is best to be idiomatic – use what fits best with your language. Follow the style of standard libraries.

Write code using the same layout conventions as the rest of your team. Don't use your own style because you think it's prettier or better. If there is no consistency on your project then consider adopting a *coding standard* or *style guide*. This does not need to be a lengthy, draconian document; just a few agreed upon layout princples to pull the team together will suffice. In this situation, coding standards must be agreed on mutually, not enforced.

If you're working in a file that doesn't follow the layout conventions of the rest of your project, follow the layout conventions in that file.

Ensure that the entire team's IDEs and source code editors are configured the same way. Get the tab stop size the same. Set the brace position and comment layout options identically. Make the line ending options match. This is particularly important on cross-platform projects where very different development environments are used simultaneously. If you aren't diligent in this, then the source code will naturally become fractured and inconsistent; you will breed bad code.

Listing 1

```
class Example
{
public:
  Example();                    // lifetime management first
  ~Example();

  void doMostImportantThing(); // this starts a new "paragraph"
  void doSomethingRelated();   // each line is like a sentence

  void somethingDifferent();   // this is another paragraph
  void aRelatedThing();

private:
  int privateStuffComesLast;
};
```

Names

We name many things: variables, functions and methods, types (e.g., enumerations, classes), namespaces, and packages. Equally important are larger things, like files, projects, and programs. Public APIs (e.g., library interfaces or web service APIs) are perhaps the most significant things we choose names for, as 'released' public APIs are most often set in stone and particularly hard to change.

A name conveys the identity of an object; it describes the thing, indicates its behaviour and intended use. A misnamed variable can be *very* confusing. A good name is descriptive, correct, and idiomatic.

You can only name something when you know *exactly* what it is. If you can't describe it clearly, or don't know what it will be used for, you simply can't name it well.

Avoid redundancy

When naming, avoid redundancy and exploit context. Consider:

```
class WidgetList {
  public int numberOfWidgets() { ... }
};
```

The `numberOfWidgets` method name is unnecessarily long, repeating the word *Widget*. This makes the code harder, and more tedious, to read. Because this method returns the size of the list, it can simply be called `size()`. There will be no confusion, as the context of the enclosing class clearly defines what `size` means in this case.

Avoid redundant words.

I once worked on a project with a class called `DataObject`. That was a masterpiece of baffling, redundant naming.

Be clear

Favour clarity over brevity. Names don't need to be short to save you key presses – you'll *read* the variable name far more times than you'll type it. But there is, however, a case for single-letter variable names: as counter variables in short loops, they tend to read clearly. Again, context matters!

Names don't need to be cryptic. The poster child for this is Hungarian Notation. It's not useful.

Baroque acronyms or 'amusing' plays on words are not helpful.

Be idiomatic

Prefer idiomatic names. Employ the capitalisation conventions most often used in your language. These are powerful conventions that you should only break with good reason. For example:

- In C, macros are usually given uppercase names.
- Capitalised names often denote types (e.g., a class), where uncapitalised names are reserved for methods and variables. This can be such a universally accepted idiom that breaking it will render your code confusing.

Be accurate

Ensure that your names are accurate. Don't call a type `WidgetSet` if it behaves like an array of widgets. The inaccurate name may cause the reader to make invalid assumptions about the behaviour or characteristics of the type.

Making yourself presentable

We come across badly formatted code all the time. Be careful how you work with it.

If you must 'tidy it up' never alter presentation at the same time as making functional changes. Check in the presentation change to source control as a separate step. *Then* alter the code's behaviour. It's confusing to see commits mixing the two things. The layout changes might mask mistakes in the functionality.

Never alter presentation and behaviour at the same time. Make them separate version-controlled changes.

Don't feel you have to pick a layout style and stick with it faithfully for your entire life. Continually gather feedback from how layout choices affect how you work with code. Learn from the code you read. Adapt your presentation style as you gain experience.

Over my career, I have slowly migrated my coding style, moving ever towards a more consistent and easier to modify layout.

From time to time, every project considers running automated layout tools over the source tree, or adding them as a pre-commit hook. This is always worth investigating, and rarely worth using. Such layout tools tend to be (understandably) simplistic, and are never able to deal with the subtleties of code structure in the real world.

Conclusion

Stop fighting about code presentation. Favour a common convention in your project, even if it's not your personal preferred layout style.

But do have an informed opinion on what constitutes a good layout style, and why. Continually learn and gain more experience from reading other code.

Strive for consistency and clarity in your code layout. ■

Questions

- Should you alter layout of legacy code to match the company coding standard? Or is it better to leave it in the author's original style? Why?
- Which is more important: good code presentation or good code design?
- How consistent is your current project's code? How can you improve this?
- Tabs or spaces? Why? Does it matter?
- Is it important to follow a language's layout and naming conventions? Or is it useful to adopt a different 'house style' so you can differentiate your application code from the standard library?
- Does our use of colourful syntax-highlighting code editors mean that there is less requirement for certain presentation concerns because the colour helps to reveal code structure?

Notes

[1] Vim is. That is all.

[2] This is not just an academic example to fill books! Serious real-life bugs stem from these kinds of mistakes. Apple's infamous 2014 *goto fail* security vulnerability in its SSL/TLS implementation was caused by exactly this kind of layout error.

Mention ACCU and receive the U.S. training rate for any location in Europe!

Live on-site C++ Training
by Leor Zolman

Courses:

Moving Up to Modern C++:
An Introduction to C++11/14/17 for experienced C++ developers. Written by Leor Zolman.
3-day, 4-day and 5-day formats.

Effective C++:
A 4-Day "Best Practices" course written by Scott Meyers, based on his Legacy C++ book series. Updated by Leor Zolman with Modern C++ facilities.

An Effective Introduction to the STL:
In-the-trenches indoctrination to the Standard Template Library. 4 days, intensive lab exercises, updated for Modern C++.

www.bdsoft.com • bdsoftcontact@gmail.com • +1.978.664.4178

On Share and Share Alike

A Student gets to grips with the Baron's coin puzzle.

When last they met, the Baron challenged Sir R----- to a wager in which, for a price of three coins and fifty cents, he would make a pile of two coins upon the table. Sir R----- was then to cast a four-sided die and the Baron would add to that pile coins numbering that upon which it settled. The Baron would then make of it as many piles of equal numbers of no fewer than two coins as he could muster and take back all but one of them for his purse. After doing so some sixteen times, Sir R----- was to have as his prize the remaining pile of coins.

The upshot of these rules is that at each turn the pile of coins would be reduced to the lowest prime factor of the number that it had after those indicated by the die were added, the primes being those positive integers that cannot be wholly divided by any positive integers other than themselves and one. For example, if Sir R----- had started a turn with seventeen coins and had thrown a one then he would have ended it with

$$17 + 1 = 18 = 3^2 \times 2 \to 2$$

coins.

The first consequence of this is that the pile could only consist of a prime number of coins at the end of a turn.

The second is that it could never be larger than twenty three coins, since

$$\begin{aligned}
23 + 1 &= 24 = 3 \times 2^3 = \left(3 \times 2^2\right) \times 2 \to 2 \\
23 + 2 &= 25 = 5^2 = 5 \times 5 \to 5 \\
23 + 3 &= 26 = 13 \times 2 \to 2 \\
23 + 4 &= 27 = 3^3 = 3^2 \times 3 \to 3
\end{aligned}$$

We need therefore only consider the implications of the rules of the wager for piles of two, three, five, seven, eleven, thirteen, seventeen, nineteen and twenty three coins in order to figure its fairness.

If we construct a table showing how many ways Sir R----- could start and end a turn with some numbers of coins in the pile by column and row respectively

		Start								
		2	**3**	**5**	**7**	**11**	**13**	**17**	**19**	**23**
	2	2	2	2	2	2	2	2	2	2
	3	1		1	1	1	1	1	1	1
	5	1	1							1
	7		1	1						
End	**11**				1					
	13					1				
	17						1			
	19							1		
	23								1	

create a matrix of its elements, and multiply it by one quarter

$$\mathbf{M} = \begin{pmatrix} 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \times \frac{1}{4}$$

then the result $\mathbf{M}$ is the transition matrix of a turn, representing the probabilities of beginning and ending it with the numbers of coins associated with the table's columns and rows.

If we further define a vector $\mathbf{v}$ with

$$\mathbf{v} = \begin{pmatrix} p_2 \\ p_3 \\ p_5 \\ p_7 \\ p_{11} \\ p_{13} \\ p_{17} \\ p_{19} \\ p_{23} \end{pmatrix}$$

where p_n is the probability that the pile contains n coins at the start of a turn, then $\mathbf{M} \times \mathbf{v}$ is a vector whose elements correspond to the probabilities that it should contain a given number of coins at the end of that turn.

That this is necessarily so follows firstly from the fact that the transitions from a particular starting number of coins to a particular ending number are independent, meaning that we may simply multiply the probabilities of the former by the probabilities of those transitions and add together the resulting probabilities of each of the latter and secondly that, by the rules of matrix-vector multiplication, we have

$$(\mathbf{M} \times \mathbf{v})_i = \sum_j \mathbf{M}_{ij} \times \mathbf{v}_j$$

where $\sum$ is the summation sign, which is trivially that sum of the products of those probabilities.

Now it must also be the case that if the probabilities of the pile having particular numbers of coins at the start of a turn are $\mathbf{M} \times \mathbf{v}$, then at the end of that turn they must be

$$\mathbf{M} \times (\mathbf{M} \times \mathbf{v}) = \mathbf{M}^2 \times \mathbf{v}$$

and so if they were instead $\mathbf{v}$ at the start of a turn, they will be $\mathbf{M}^2 \times \mathbf{v}$ at the end of the following turn, since the rolls of the die at each turn are also independent.

Carrying on in the same fashion, we find that if the probabilities were $\mathbf{v}$ at the start of the game, then they should be $\mathbf{M}^{16} \times \mathbf{v}$ at its conclusion.

Such probabilistic systems are known as Markov chains and are of tremendous utility when it comes to figuring the probabilities of the outcomes of processes for which the transitions between various states at any given time are independent of those that have come before, not least for the ease with which we may use matrices and vectors to do so. I said as much to the Baron, but I fear that I may not have had his full attention.

Now we may spare ourselves some considerable effort by noting that

$$\begin{aligned}\mathbf{M} \times \mathbf{M} &= \mathbf{M}^2\\ \mathbf{M}^2 \times \mathbf{M}^2 &= \mathbf{M}^4\\ \mathbf{M}^4 \times \mathbf{M}^4 &= \mathbf{M}^8\\ \mathbf{M}^8 \times \mathbf{M}^8 &= \mathbf{M}^{16}\end{aligned}$$

and so we need but four matrix multiplications to figure the probabilities of the pile having any particular number of coins at the game's end. With some small measure of patience my fellow students and I have reckoned these to equal the equation in Figure 1.

Since the probability of having two coins in the pile at the outset equals one, the initial probability vector must have a one for its first element and zero for the rest and so the product of a matrix and it is simply equal to the first column of that matrix, giving us a probability vector at the end of the game of

$$\begin{pmatrix}p_2\\p_3\\p_5\\p_7\\p_{11}\\p_{13}\\p_{17}\\p_{19}\\p_{23}\end{pmatrix} = \mathbf{M}^{16} \times \begin{pmatrix}1\\0\\0\\0\\0\\0\\0\\0\\0\end{pmatrix} = \begin{pmatrix}2,147,483,648\\858,993,459\\751,717,587\\402,677,762\\100,669,440\\25,167,360\\6,291,840\\1,572,960\\393,240\end{pmatrix} \times \frac{1}{4^{16}}$$

Finally, Sir R-----'s expected prize was simply

$$p_2 \times 2 + p_3 \times 3 + p_5 \times 5 + p_7 \times 7 + p_{11} \times 11$$
$$+ p_{13} \times 13 + p_{17} \times 17 + p_{19} \times 19 + p_{23} \times 23$$

which comes out to

$$\frac{7,514,855,751}{2,147,483,648} = 3 + \frac{1,072,404,807}{2,147,483,648}$$

Now the cost of the wager was

$$3 + \frac{50}{100} = 3 + \frac{1,073,741,824}{2,147,483,648}$$

and so this represents a slight loss for Sir R----- and I should have advised him to decline it. □

Acknowledgement

Courtesy of www.thusspakeak.com

Figure 1

$$\mathbf{M}^2 = \begin{pmatrix}8&8&8&8&8&8&8&8&8\\3&4&3&3&3&3&3&3&3\\3&2&3&3&3&3&3&4&3\\2&1&1&1&1&1&1&1&2\\0&1&1&0&0&0&0&0&0\\0&0&0&1&0&0&0&0&0\\0&0&0&0&1&0&0&0&0\\0&0&0&0&0&1&0&0&0\\0&0&0&0&0&0&1&0&0\end{pmatrix} \times \frac{1}{4^2}$$

$$\mathbf{M}^4 = \begin{pmatrix}128&128&128&128&128&128&128&128&128\\51&52&51&51&51&51&51&51&51\\45&44&45&45&45&46&45&45&45\\24&24&24&24&24&24&25&24&24\\6&6&6&6&6&6&6&7&6\\2&1&1&1&1&1&1&1&2\\0&1&1&0&0&0&0&0&0\\0&0&0&1&0&0&0&0&0\\0&0&0&0&1&0&0&0&0\end{pmatrix} \times \frac{1}{4^4}$$

$$\mathbf{M}^8 = \begin{pmatrix}32,768&32,768&32,768&32,768&32,768&32,768&32,768&32,768&32,768\\13,107&13,108&13,107&13,107&13,107&13,107&13,107&13,107&13,107\\11,471&11,469&11,470&11,470&11,470&11,470&11,470&11,470&11,471\\6,144&6,145&6,145&6,144&6,144&6,144&6,144&6,144&6,144\\1,536&1,536&1,536&1,537&1,536&1,536&1,536&1,536&1,536\\384&384&384&384&385&384&384&384&384\\96&96&96&96&96&97&96&96&96\\24&24&24&24&24&24&25&24&24\\6&6&6&6&6&6&6&7&6\end{pmatrix} \times \frac{1}{4^8}$$

$$\mathbf{M}^{16} = \begin{pmatrix}2,147,483,648&2,147,483,648&2,147,483,648&2,147,483,648&\ldots\\858,993,459&858,993,460&858,993,459&858,993,459&\ldots\\751,717,587&751,717,586&751,717,587&751,717,587&\ldots\\402,677,762&402,677,761&402,677,761&402,677,761&\ldots\\100,669,440&100,669,441&100,669,441&100,669,440&\ldots\\25,167,360&25,167,360&25,167,360&25,167,361&\ldots\\6,291,840&6,291,840&6,291,840&6,291,840&\ldots\\1,572,960&1,572,960&1,572,960&1,572,960&\ldots\\393,240&393,240&393,240&393,240&\ldots\end{pmatrix} \times \frac{1}{4^{16}}$$

A Brief Introduction to Docker

Silas S. Brown shares his experiences of setting up a virtual appliance.

Docker is basically a convenient way of setting up chroot jails on the GNU/Linux platform, but some companies now use it to deploy software to their servers. Docker is like having a lightweight virtual machine, except only on Linux (don't expect to be able to run it on Windows or Mac except inside a Linux VM). One advantage of Docker over virtual machines is ease of initial setup. For example, on several versions of Red Hat Linux as used in some corporate environments, Virtualbox won't run without significant extra effort, but Docker 'just works'. Want to do something on a virtual Debian box? Install Docker, ensure **`sudo dockerd`** is running, and do:

```
docker pull debian
docker run -it debian /bin/bash
```

and you should be away (except you'll need an **`apt-get update`** before doing any serious amount of package installation).

But as soon as that first shell exits, any changes you made to the system (such as installing extra packages and changing the configuration) will be lost. That might be OK for a one-off experiment, but for serious use you probably want some of the configuration to persist. That's usually done by writing build instructions for your own derived Docker image.

Just as the **`make`** command uses a file typically named Makefile, so Docker uses a file called Dockerfile, which should be placed at the top of your source tree (or at least the part of it relevant to the Docker image you're creating). Dockerfiles almost always name an existing GNU/Linux distribution to use as a base, followed by files or directories to copy into the container and setup commands to run:

```
FROM centos:6.8
COPY myDirectory /etc/myDirectory
COPY src/*.c /home/user/src/
RUN yum install -y myPackage
```

but if you need to start daemons, you shouldn't do so as an effect of the **`RUN`** commands here, since these run only when the image is generated, not every time it's started. You may add a single **`CMD`** command to the Dockerfile saying what command should be run when **`docker run`** is called on the image (there's also an alternative called **`ENTRYPOINT`** which can take additional command-line arguments from the **`docker run`** command, in which case **`CMD`** is repurposed to specify default arguments to add when these are missing), and this will be responsible for starting any necessary daemons, running the foreground process, and, if necessary, cleanly shutting down the daemons afterwards (otherwise they'll all be aborted as soon as the master process exits).

It may also be worth noting that Docker will try to cache all intermediate states between commands in the Dockerfile, so combining multiple **`RUN`** commands into one can save disk space. (**`RUN`** commands are also expected to produce practically-identical results each time they are run: the cache will be refreshed if the source file of a **`COPY`** is changed, but it will not be refreshed just because the expected result of some **`RUN`** command changes unless the **`RUN`** command itself is changed. This might affect you if you try to install your own work via a network-fetching **`RUN`** instead of via a **`COPY`**: changes you make upstream will not be reflected in the Docker build, unless you give Docker some other reason to invalidate its cache before reaching that **`RUN`**, such as by making changes to a file that's **`COPY`**'d in first.)

Base images can be found on https://hub.docker.com/explore/ but the presence of application-specific base images (nginx, golang etc) is slightly misleading: you can't 'import' multiple applications by depending on multiple base images, so it's not as much of a 'package manager' as it seems. Granted, if you need one base environment to compile something, but then wish to copy only its final binary into another base environment (discarding the compiler etc), you can do this with Docker's 'multi-stage builds':

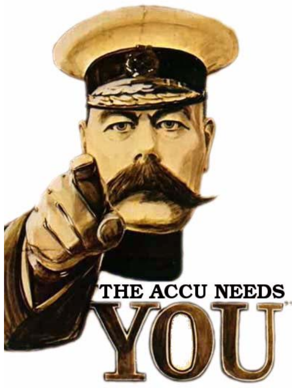
```
FROM golang:1.7.3 as builder
RUN build-my-Go-program
FROM centos:6.8
COPY --from=builder /path/to/my/binary .
```

but you'd then have to make sure all the right libraries are in the final image. This might be useful if you need a compiler that's harder to set up on all development machines due to distribution differences, and don't want the bloat of putting the compiler in the final image. But beyond this, there unfortunately doesn't yet seem to be a way of asking Docker to 'merge' base images, so you can't use Docker itself as a package manager. At least it makes it easier to obtain minimal distributions (which can be different from the distribution you're running) and use their own package managers. Please change the distro's package-manager configuration to use your nearest mirror before downloading large packages with it, especially in a Docker image that's likely to be re-built frequently (in extreme cases it might even be a good idea to cache some packages locally).

Further documentation can be found on docker.com; the **`EXPOSE`** command is worth a look if you want to run a server inside the container that you wish to be visible from the outside. ■

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



Write for us!

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

What do you have to contribute?

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

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

ACCU – The Early Days (Part 1)

Francis Glassborow recalls how the ACCU came about.

Back when my career as a teacher was coming to a close (stress-related ill-health was the proximate cause, but that is another story), in June of that same year, I came across a small ad in a news-stand magazine (*Computer World*, if memory serves me correctly) for the C User Group (UK). I thought it might be interesting and sent off my £10 for membership and six issues of its newsletter. I soon received a mimeographed A4 newsletter titled *C Vu*, which largely consisted of reprints from the US based CUG (long deceased, and which continued as a ghost organisation for a number of years having been absorbed by its newsletter, *The C Users Journal*, which had grown up to being a glossy print publication). By the way, try searching the Web for 'C magazines'; not at all the results you might expect. In addition, the Wikipedia entry is completely wrong, not least because both *CVu* and *Overload* continue as print magazines. Anyone got the time to get them to correct the entry?

After several months, the next issue had not materialised on my doormat so, before actually giving up and assuming that it was a transient publication, I rang the contact number for Martin Houston that was on the issue I had (I think it was issue 4) to ask what had happened to the next issue. He told me that there wasn't one yet, but that my telephone call was very timely as he was planning a meeting to discuss the future of CUG(UK) for January. He had booked a room at the Aeronautical Institute in London.

I decided that I would go and that was one of those momentous decisions that seem so unimportant when they are made. I arrived in a very luxurious room to find a couple of dozen other C enthusiasts.

The discussion that ensued began to meander. Being an old hand at taking control of meetings, I looked around and suggested that we had just about what we needed for a Committee. Martin Houston was obviously the person for the chair and I volunteered to be membership secretary (well, it seemed a pretty harmless job). Someone volunteered as secretary and someone else as editor. I am afraid that I have forgotten the names, though I guess a bit of digging could bring them to light.

I next addressed the issue of regular publication. I pointed out that people would only write articles in a timely fashion if they had deadlines and a regular publication schedule. The original decision was for it to be quarterly. As we will see, that was not to last very long.

The meeting closed and I started find computer-interest events that I could beg a free table at to publicise CUG(UK). That brought me in contact with a lot of interesting people, including Alan Lenton. It also resulted in my being invited to a lunch with the person responsible for the computer books at Addison Wesley. Out of that contact grew our wide reviewing coverage of programming books. I can still recall a conversation at one show with the representative for Wiley's. She made some remark about editors who just asked for review copies of books because they wanted them for their personal bookshelves. I replied that, in all honesty, I did the same. I can remember the smile on her face when she told me that I was not like those other editors because, unlike them, I read and reviewed the books before putting them on my shelf.

Enough for this episode. Next time (if the editor likes the idea) I will tell you how I came to be editor of *CVu* and how I had the nerve to tell Bjarne Stroustrup that the 2nd edition of *The C++ Programming Language* was much better than the first. ■

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++*.

Good Intentions

R. Brian Clark proves that sometimes tomorrow does eventually come.

The road to hell is paved with good intentions.

Never do today what you can put off 'til tomorrow.

I've had good intentions for a long time to write something for an ACCU journal but kept putting it off until tomorrow! However, the request from the Membership Secretary to give details of my Honorary Membership has at last roused me to pen the following. Please don't be put off by what is now considered to be somewhat old fashioned jargon.

I can't remember when I joined CUG(UK), the forerunner of ACCU, but it was a few years after its formation. Francis Glassborow kept requesting articles for the journal and at last I've got round to submitting one! He also kept asking me to attend the AGM, which then I think was held at a Motorcycle Museum in the Birmingham area. As they were held on a Saturday, I never did, as the weekends were the only time I had to devote full time to my family.

In 1998, I retired and the ACCU very kindly offered me Honorary Membership. In the early years, email wasn't as widely available as now and some members of the Committee weren't connected! So I acted as a hub for email, forwarding minutes of meetings, messages, etc to those who had and by smail to those who hadn't, and also helped in a minor administrative role. Then my email address was username@uk.man.ac and you had to route the messages, eg to send to the USA it went via University College, London. However, we thought it was wonderful as it was so much faster than smail and cheaper than a telephone call. You had to be careful not to use too much bandwidth. Of course, by 1998 I had long stopped doing these as everyone was connected and we had www (who remembers when it was text driven!).

It may be of interest to give a brief account of my life in computing from those far off days. In 1978, I was appointed 'Computer Manager' to *manage* a DEC machine which the Department had just bought. My knowledge of computing then was as an infrequent user but I knew a little about what made them tick. Some years previously, I had to do some calculations and it was suggested that I did them on the University computer. So I learnt Atlas Autocode – sorry, I can't now remember anything about it. In preparation for my new job I learnt Basic, and some Assembly Language. About 1980, one of the professors said he had heard that Unix developed by Bell Telephones was a much better operating system then DEC's so off I went to Newcastle University with a removable Winchester Disc and got a copy. It was duly installed and over the years updated, as was the computer.

The Unix machine was considered as a main frame, so very early on Apples and BBC mini computers were also used and eventually computers running Windows. I used to write programs in Fortran and Pascal and finally C. A few years before I retired, I attended a course on C++ given by B Stroustrup and as it seemed a bit complicated and C fulfilled my requirements I decided not to bother.

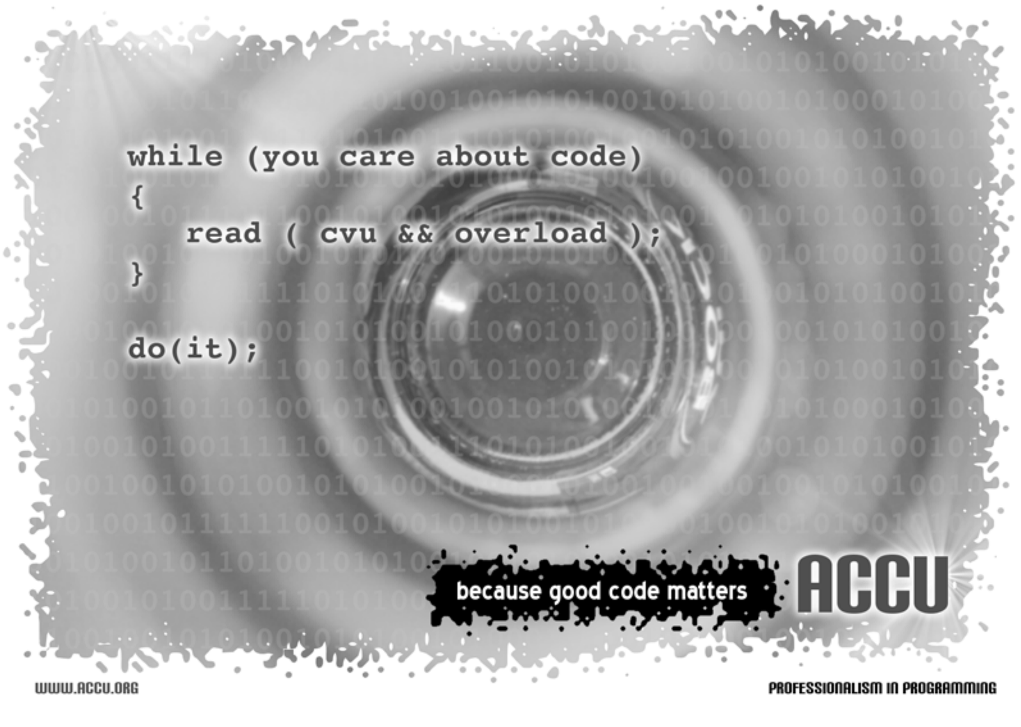
When I retired, I installed Linux, which I still use, on my PC with the intention of installing some of the programs I had written. Of course, I only did a couple before my interest waned and I started on something else.

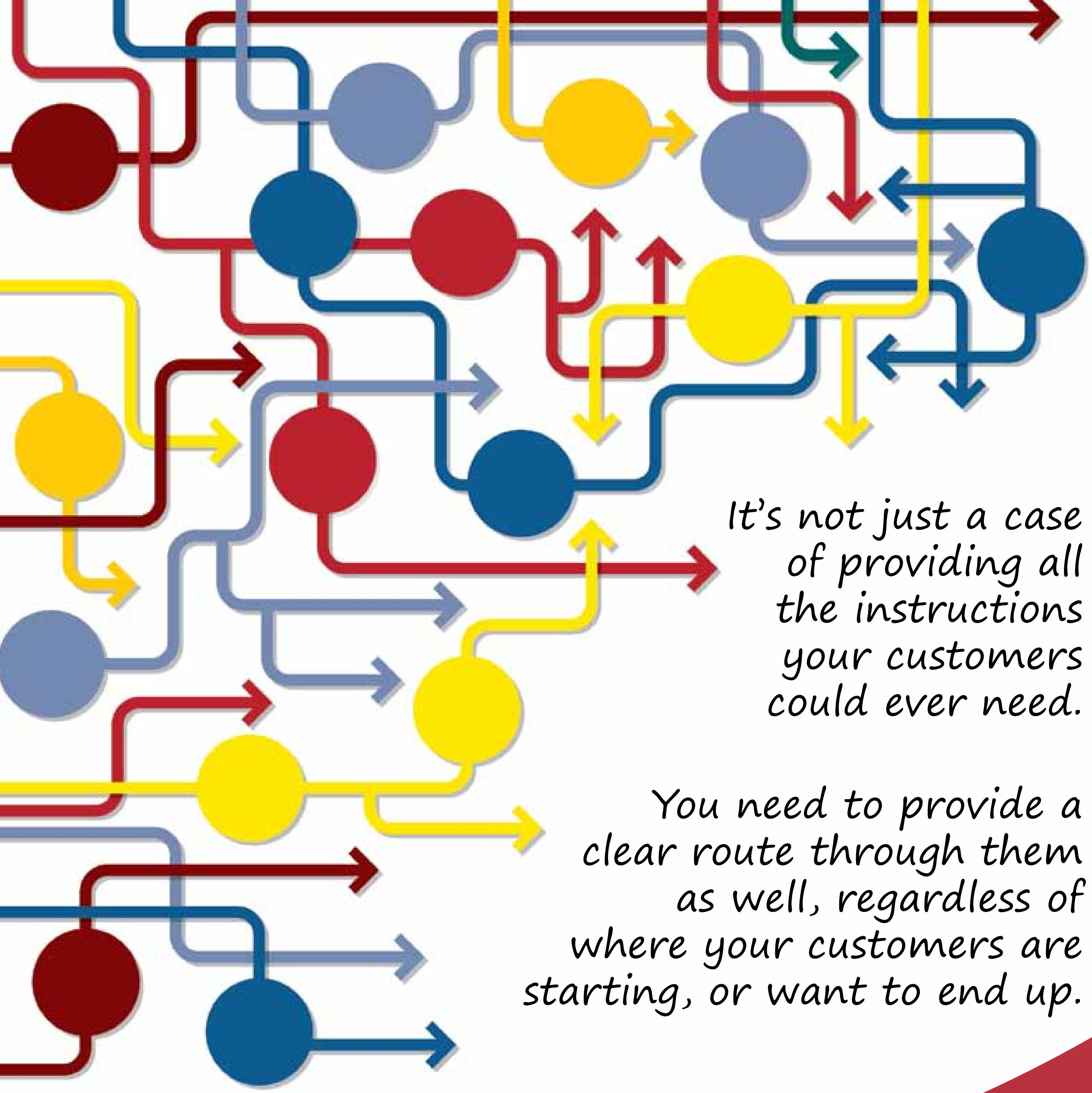
I started computing untrained, I wonder would I be taken on now with such a background?

I hope you have found the above interesting and don't consider them the ramblings of an old man. ■

R. BRIAN CLARK

R. Brian Clark is an Honorary Member of ACCU and has recently been contacted by the Membership Secretary as part of the project to document the association's history.


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



It's not just a case of providing all the instructions your customers could ever need.

You need to provide a clear route through them as well, regardless of where your customers are starting, or want to end up.

Get in touch for an informal discussion on how we can help you:

Clearly stated...

T 0115 8492271

E info@clearly-stated.co.uk

W www.clearly-stated.co.uk

Two Pence Worth

Another crop of sage sayings from ACCU members.

One of the marvellous things about being part of an organisation like ACCU is that people are always willing to help out and put in their two-pence-worth of advice. We are capturing some of those gems and will print the best ones.

"Write your documentation in LaTeX in case a customer wants a bound copy of it." P. Diephe, UK

"Embrace cultural diversity in your team by using Unicode variable names." Anon

"Rebuild your machine every week to make sure you don't pick up spurious DLLs when testing." James B, London

"Managers: encourage your coders to work on Open Source projects, giving you free access to any good ideas they have outside of work hours." A Consultant, Chicago

"Lose weight by ensuring your code doesn't cause a build failure when you check it in." S Dahli, UK

"Make sure your developers are using the correct tab/space policy by reviewing their code changes with Notepad." T. More, Canterbury

"Before editing every source file create a new branch so that you can merge the required features into your release branch." N Machiavelli, Italy

"Improve compilation times by removing all whitespace from your source files." Phil H, Birmingham

"Give your code a more mathematical feel by overloading lots of different operators." Anon

"Save time on lengthy handover notes when you leave by documenting as you go along." Anon

"Buy your manager two copies of *The Mythical Man Month* so they can read it quicker." Fred B, USA

"Avoid having to keep learning new programming languages by just transpiling your favourite language to JavaScript." Ryan D, The Internet

If you have your own 2p to add, send it to cvu@accu.org

Old money, new money...

Alison Peck looks at the origins of 'two pence worth'.

I can still hear adults in my local area, in an exasperated tone of voice after finally losing patience with children 'butting in', saying, "You just had to add your tupp'orth, didn't you?" I hadn't thought about the phrase for a long time – not until we did the first in this (very occasional) series back in 2012 – and it set me thinking. What exactly *does* it mean?

Basically, it has to do with money. Pre-decimal British money, to be precise. As now, you could get a penny coin and a two-penny coin. (As an aside, the names of the coins then sounded much more interesting than now to my ears – half-a-crown, a sixpence, a thrupenny bit… and don't get me started on guineas!)

Almost instinctively, I equated tuppence (two pennies) to 'a small amount', but one still worth counting. I can (vaguely) remember going to the corner shop to buy tuppence – or sometimes *thruppence* (three pennies) – worth of sweets with my pocket money, which was a threepenny (pronounced 'thrup'ny') bit. I stress I was *very* young at the time! ☺

But am I right? After a bit of research online, the answer is, *probably*. Most of the sources indicate that there are very similar phrases in most of the English-speaking world, with variations to account for currency differences. For example, Wikipedia suggests that 'My 2¢' was first used in print in March 1926 [1] as the title of a newspaper article in the USA.

The current usage is more to do with self-deprecation before joining in a discussion, or immediately before or after offering an opinion. However, this is starting to remove the 'but still worth counting' part of my own understanding, and concentrating on the (potential) low worth.

Explanations for the origins of the phrase fit well with my British upbringing and my personal understanding of its meaning. First, 'A penny for your thoughts' is another familiar phrase – offering to pay a penny (a token amount) but being given twice as much (tuppence worth) suggests you got more than you were expecting, and maybe more than you wanted in return. [2] Secondly, it seems that the phrase started to be used in the UK in the middle of the 19th Century, and there is a suggested link with the standard cost of sending a letter: tuppence. When I get 5 minutes, I may investigate further…

Oh, and if you're interested in the English language – its history and evolution, its spelling and its regional variations – I suggest you read some of David Crystal's excellent books on a fascinating subject. [3]

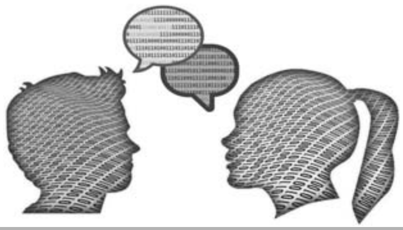
References

[1] https://en.wikipedia.org/wiki/My_two_cents
[2] https://idioms.thefreedictionary.com/put+in+my+tuppence+worth
[3] http://www.davidcrystal.com/biography

ALISON PECK
Alison loves language and the ways we communicate. She couldn't write a novel to save her life, so instead puts her enthusiasm into her work as a technical author and trainer. Contact her at alison@clearly-stated.co.uk

Code Critique Competition 108

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



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

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

Last issue's code

I want to collect the meals needed for attendees for a one-day event so I'm reading lines of text with the name and a list of the meals needed, and then writing the totals. However, the totals are wrong – but I can't see why:

```
> meals
Roger breakfast lunch
John lunch dinner
Peter dinner

Total: 3 breakfast: 3 lunch: 2 dinner: 2
```

There should only be 1 breakfast, not 3!

Please can you help the programmer find the bug – and suggest some possible improvements to the program!

- Listing 1 contains `meal.h`
- Listing 2 (overleaf) contains `meals.cpp`

Listing 1

```
#pragma once
#include <iosfwd>
#include <sstream>
#include <string>
enum class meal : int
{
   breakfast, lunch, dinner,
};

// Used for name <=> value conversion
struct
{
  meal value;
  std::string name;
} names[] =
{
  { meal::breakfast, "breakfast" },
  { meal::lunch, "lunch" },
  { meal::dinner, "dinner" },
};

std::istream &operator>>(std::istream &is,
  meal &m)
{
  std::string name;
  if (is >> name)
  {
    for (auto p : names)
    {
      if (p.name == name)
        m = p.value;
    }
```

Listing 1 (cont'd)

```
  }  return is;
}

std::ostream &operator<<(std::ostream &os,
  meal const m)
{
  for (auto p : names)
  {
    if (p.value == m)
      os << p.name;
  }
  return os;
}

// Type-safe operations
constexpr meal operator+(meal a, meal b)
{
  return meal(int(a) + int(b));
}

meal operator+=(meal &a, meal b)
{
  a = a + b;
  return a;
}

constexpr meal operator|(meal a, meal b)
{
  return meal(int(a) | int(b));
}

constexpr meal operator&(meal a, meal b)
{
  return meal(int(a) & int(b));
}

// Check distinctness
static_assert((meal::breakfast | meal::lunch |
 meal::dinner) == (meal::breakfast +
 meal::lunch + meal::dinner), "not distinct");
```

Critiques

Kaartic Sivaraam <kaarticsivaraam91196@gmail.com>

The issue seems to be with the 'enum class'

```
enum class meal : int
{
  breakfast, lunch, dinner,
};
```

They have the values of 0, 1, 2 each having 'at most' one bit set in binary, of course. The issue is that as 'breakfast' has value of 0 it is impossible to

ROGER ORR
Roger has been programming for over 20 years, most recently in C++ and Java for various investment banks in Canary Wharf and the City. He joined ACCU in 1999 and the BSI C++ panel in 2002. He may be contacted at rogero@howzatt.demon.co.uk

Listing 2

```
#include "meal.h"
#include <iostream>
#include <list>
struct attendee
{
   std::string name;
   meal meals; // set of meals
};

using attendees = std::list<attendee>;

attendees get_attendees(std::istream &is)
{
  attendees result;
  std::string line;
  while (std::getline(is, line))
  {
    std::istringstream iss(line);
    std::string name;
    iss >> name;
    meal meal, meals{};
    while (iss >> meal)
      meals += meal; // add in each meal
    if (is.fail())
      throw std::runtime_error("Input error");
    result.push_back({name, meals});
  }
  return result;
}
```

distinguish between the one who wants breakfast and who doesn't. That's because the expression

```
(item.meals & m) == m
```

always evaluates to **`true`** for the value of **`breakfast`** (0). The following change would fix the issue.

```
enum class meal : int
{
  breakfast=1, lunch=2, dinner=4,
};
```

Further, the following method seems to need a little tweak,

```
std::istream &operator>>(std::istream &is,
  meal &m)
```

It doesn't seem to be setting the **`failbit`** of the **`is`** in case it doesn't identify the 'valid' input. I guess the following change fixes that. Correct me, if I'm wrong.

```
std::istream &operator>>(std::istream &is,
  meal &m)
{
  std::string name;
  bool found = false;
  if (is >> name)
  {
    for (auto p : names)
    {
      if (p.name == name) {
        m = p.value;
        found = true;
        break;
      }
    }
  }
  if(!found)
    is.setstate(std::ios::failbit);
  return is;
}
```

Nitpicking a little more, the file `meal.h` doesn't look like a header though it has the `.h` suffix, which isn't such a good thing. It should be split into two files named `meal.h` and `meal.cpp`. The file currently named `meals.cpp` should be renamed to something else more meaningful, probably `attendee_meal_requirements.cpp`.

Russel Winder <russel@winder.org.uk>

Having looked at the C++ code presented for Code Critique 107, my reaction was WTF. After a moment to collect myself, I realised: C++ is just totally the wrong language with which to attempt the problem as stated. Thus I provide no answer to "Please can you help the programmer find his bug" (*), and thus likely relinquish any possibility of winning this code critique. However for "suggest some other possible problems with their program." I suggest: the single biggest problem with this code is that it is written in C++.

Clearly I need to follow up on this conclusion with some positive and constructive thoughts.

What language would be good for this problem? The answer is clearly (**) Python (https://www.python.org). In `rw_meals.py` I provide a program that answers the problem as set by "I". Python has proper enums and sets, so obviates the need for all the extra code to be found in the C++ files: "I" had some good implementation ideas, but trying to realise them in C++ is, well let us be kind and say non-trivial. Multiple calls of the C++ count function are replaced by a single call of the Python reduce function from the functools package (Python 3 is, of course, assumed here). The brevity and clarity of the Python code mean that it is likely right just by observation. However, given the test data:

```
|> rw_meals.py
   Roger breakfast lunch
   John lunch dinner
   Peter dinner
   Total: 3, breakfast: 1, lunch: 2, dinner: 2
```

we discover the code actually does give the answer expected. Obviously though this is but a single functional test, more tests, unit as well as functional should be provided if the code is to go into service for more than this one occasion.

Of course there will be people unwilling to accept this Python code as a solution because it isn't native code, or some other third rate excuse of this sort. OK, I shall cope with this quasi-objection by providing a solution using D (https://dlang.org). In `rw_meals.d` is code that is very similar to the Python code, and produces the same result, the correct result:

```
|> rdmd rw_meals.d
   Roger breakfast lunch
   John lunch dinner
   Peter dinner
   Total: 3, breakfast: 1, lunch: 2, dinner: 2
```

D doesn't have a built in set type, nor does it have an explicit set type in the standard library. The obvious (***) choices for implementing a set in D are to use a Red-Black Tree (there is an implementation in the standard library) or use an associative array (aka hash map, dictionary). In this case, I have used an associative array with the value of each key being the same as the key, which is about as close to a hash set as makes little difference.

I wonder what a good C++ code would look like? I am not sure I actually care as I have Python and D versions that satisfy me. Use the right tool for the right job as the saying goes.

Notes

(*) Let us assume "I" is actually a man, rather than this being bad phrasing by the CVu team.

(**) Though obviously others will say Ruby, or Lisp, or… well anything other than C++ really.

(***) To me, others may have a different view.

Source code follows

(Ed: reformatted slightly for publication.)

```
---- rw_meals.py ----
#!/usr/bin/env python3

import enum
import functools
import sys

class Meal(enum.Enum):
  breakfast = 'breakfast'
  lunch = 'lunch'
  dinner = 'dinner'

def get_attenders():
  result = {}
  for line in sys.stdin.readlines():
    data = line.strip().split()
    result[data[0]] =\
      {Meal(x) for x in data[1:]}
  return result

if __name__ == '__main__':
  attenders = get_attenders()
  counts = functools.reduce(
    lambda t, x: [t[0] +\
     (1 if Meal.breakfast in x else 0),
       t[1] + (1 if Meal.lunch in x else 0),
       t[2] + (1 if Meal.dinner in x else 0)],
        attenders.values(),
        [0, 0, 0]
  )
  print('Total: {}, breakfast: {}, lunch: {},\
   dinner: {}'.format(len(attenders),\
     *counts))

---- rw_meals.d ----
import std.algorithm: map, reduce;
import std.array: array, split;
import std.conv: to;
import std.stdio: lines, stdin, writefln,
  writeln;
import std.string: strip;

enum Meal: string {
  breakfast = "breakfast",
  lunch = "lunch",
  dinner = "dinner",
}
auto getAttenders() {
  Meal[Meal][string] people;
  foreach (string line; stdin.lines()) {
    auto data = line.strip().split();
    Meal[Meal] mealSet;
    foreach (item;
      data[1..$].map!(a => to!Meal(a))) {
      mealSet[item] = item;
    }
    people[data[0]] = mealSet;
  }
  return people;
}
void main() {
  auto attenders = getAttenders();
  auto counts = reduce!(
    (t, x) => [t[0] + ((Meal.breakfast in x) ?
                       1 : 0),
               t[1] + ((Meal.lunch in x) ?
                       1 : 0),
               t[2] + ((Meal.dinner in x) ?
                       1 : 0)])(
                 [0, 0, 0],
                 attenders.byValue().array);
  writefln("Total: %d, breakfast: %d,"
    " lunch: %d, dinner: %d",
    attenders.length, counts[0], counts[1],
    counts[2]);
}
```

Jim Segrave <jes@j-e-s.net>

The basic problem with this program is that the enums being used have the values breakfast: 0, lunch: 1 and dinner: 2. For every attendee, the test to see if they want breakfast tests:

```
if((item.meals & 0) == 0)
```

which will always be true, so everyone is listed as wanting breakfast. This could be fixed by changing the enum definition to read:

```
breakfast = 1, lunch = 2, dinner = 4,
```

That still leaves problems: if someone orders breakfast twice on one line, they'll get lunch instead, two lunches results in only dinner, etc. No error report is generated.

If someone orders two dinners, they only get the default breakfast, as the sum of two `dinner` enums is 4 and only 0, 1 and 2 are recognised in this code.

If one of the input lines is duplicated, that attendee will be scheduled to have two of each meal they've selected.

Invalid meal names are ignored, but not reported, which is probably not a good idea.

If someone is attending but doesn't choose any meal at all, with the current code she still gets put down for breakfast, which is wrong, if the enum is fixed, she's not considered as attending and the number of attendees will only be the number of people having a meal, which should be separate counts.

Then there are style problems:

`meal` is a class, but it's also the name of a variable, `attendees` is a class and again, the name of a variable, resulting in lines such as:

```
meal meal, meals{};
```

and

```
auto attendees{ get_attendees( std::cin) };
```

These are syntactically valid, but make reading the code difficult.

While it's valid to replace the body of `main` with a `try{} catch{}` block, it is, to say the least, not idiomatic.

The code uses function-style casts, the widespread consensus among C++ developers is that these and C-style casts should be replaced with the C++ cast operator.

The fact that no less than 4 operator overloads are used to perform dubious operations on enums (assigning the sum of two enums to an enum may be correct, but more often is not. The fact that these overloads are needed for handling enums should be a hint that an enum probably isn't the right type to use.)

Realistically, C++ isn't the right language for a task like this, an ordinary scripting language would be more appropriate (shorter, quicker to write, easier to debug). There are still other lurking issues – the attendee names and meal names are case-sensitive. While it would be easy to ensure that the meal choices are handled caselessly, names are a different problem; they may contain UTF characters or they may not be convertible between lower and upper case (I believe that there is a variant of 'I' in Turkish which has no lower case). The program also doesn't address names well in that it isn't designed to handle multi-word names (Gerrit Jan for example is often taken as a first name, not a first and middle name). The sample input uses first names only, for any sizeable list, the probability of a clash is not insignificant.

These problems require a lot more design and specification before trying to implement a real solution.

Nonetheless, I've re-written this in C++11 or later and addressed the earlier problems I've noted with duplicated entries on the same line, misspelled meal names, attendees choices being spread over more than one line. As a side benefit, it's possible to get a list of all the attendees, whether they are on a strict diet or not and, for each meal type, a list of who has chosen that meal. The program can either receive the list of attendees and their meal choices on stdin or read it from a file given as the sole argument to the program.

The meal choices have been moved to an initializer list at the top of the C++ file, simply adding another meal name to that list is sufficient to enable it to be processed (note the commented out **"high-tea"** in the initializer list).

I've attached the header file `meal.h`, the C++ code `cc106.cpp`, and a somewhat larger sample input file `cc106.input`.

```
-- revised header file --
#pragma once
#include <fstream>
#include <initializer_list>
#include <iostream>
#include <iomanip>
#include <sstream>
#include <map>
#include <string>
#include <vector>
class Meal_name {
 private:
    static int        new_id;
    static int        total_meals;
    int               count;
 public:
    int         const id;
    std::string const meal_name;
    Meal_name(const char * name);
    // return number of these meals needed
    int  get_count() const { return count; }
    // return total number of meals
    // (all types)
    int  get_total_meals() const {
      return total_meals; }
    // somebody wants one of these meals
    void incr() { ++count; ++total_meals;}
};
using Selection    =
  std::pair<const std::string, int>;
using Attendee_map =
  std::map<const std::string, int>;
using Choice_vec   =
  std::vector<Meal_name>;

ssize_t lookup_meal(const std::string & name,
  const Choice_vec & vec);

Selection parse_line(
  const std::string & input_line, int line_no,
  Attendee_map & attendees,
  const Choice_vec & vec);

void process_line(
  const std::string & input_line, int line_no,
  Attendee_map & attendees,
  Choice_vec & vec);
```

```
-- revised C++ code --
#include "meal.h"
// initializer list with the names of all the
// different meals attendees can choose
std::initializer_list<Meal_name> all_choices
{"breakfast", "lunch", "dinner",
 /* "high-tea", */ };
// ensure each Meal_name has an id which is a
// power of 2 (1, 2, 4, ...)
int Meal_name::new_id = 1;
Meal_name::Meal_name(const char * name)
: count(0),
  id(Meal_name::new_id), meal_name(name) {
  new_id *= 2;
}
int Meal_name::total_meals = 0;

// return index of meal 'name' in the vector
// or -1 if not found
ssize_t lookup_meal(const std::string & name,
  const Choice_vec & vec) {
  for(size_t i = 0; i <  vec.size(); ++i) {
    if(vec[i].meal_name.compare(name) == 0) {
      return static_cast<ssize_t> (i);
    }
  }
  return -1;
}
// parse an input line, returning the attendee
// name (if any) and the bitwise
// or of the meals selected (blank lines will
// return this as zero)
Selection parse_line(
  const std::string & input_line, int line_no,
  Attendee_map & attendees,
  const Choice_vec & vec) {

  std::string    name;
  std::istringstream is(input_line);
  // skip blank lines
  is >> name;
  if(name.size() == 0) {
    // skip blank lines
    return Selection{"", 0};
  }
  // ensure the name is registered if new
  attendees.emplace(name, 0);
  int choice = 0;
  std::string meal;
  // accumulate all errors for this input line
  std::ostringstream errs;
  while(is >> meal) {
    auto idx = lookup_meal(meal, vec);
    if(idx < 0) {
      errs << "\t\"" << meal <<
        "\" is not a valid name for a meal" <<
        std::endl;
      continue;
    }
    if((choice & vec[idx].id) != 0) {
      errs << "\t\"" << meal <<
        "\" appears more than once"
        " on this line" << std::endl;
      continue;
    }
    choice |= vec[idx].id;
  }
  // report if there were any problems on this
  // line
  if(errs.str().size() != 0) {
    std::cout << "Line " << line_no << ": "
      << input_line << std::endl;
    std::cout << errs.str();
  }
  return Selection(name, choice);
}
void process_line(
  const std::string & input_line, int line_no,
  Attendee_map & attendees, Choice_vec & vec) {
  Selection sel(parse_line(
    input_line, line_no, attendees, vec));
  const std::string & name = sel.first;
  int choice = sel.second;
  if(choice == 0) {
    // blank line or no meals ordered on this
    // line
    return;
  }
  // previously ordered meals
  int old_choice = attendees[name];
  // newly ordered meals
  int new_choice = ~old_choice & choice;
  int idx = 0;
  while(new_choice != 0) {
    if(new_choice & 1) {
      // update no. of these meals ordered,
      // total meals
      vec[idx].incr();
    }
    new_choice >>= 1;
    ++idx;
  }
  attendees[name] = choice | old_choice;
}
int main(int argc, char **argv) {
  // set up the possible meals
  Choice_vec meal_choices(all_choices);
  // a map to track attendees
  Attendee_map attendees;
  // use cin unless there's a CLI parameter
  // (file of input lines)
  std::istream *f = &std::cin;
  std::ifstream ifs;
  ifs.exceptions(std::ifstream::badbit |
    std::ifstream::failbit);
  try {
    if(argc > 1) {
      ifs.open(argv[1]);
      f = &ifs;
    }
    std::string line;
    int     line_no = 0;
    while(std::getline(*f, line)) {
      process_line(line, ++line_no,
        attendees, meal_choices);
      if(ifs.eof()) {
        ifs.close();
        break;
      }
    }
  }
  catch (std::ios_base::failure &ex) {
    std::cerr << "File read error on " <<
     ((argc > 1) ? argv[1] : "standard input")
     << std::endl;
    exit(1);
  }
  std::cout << "\nAttendees: " <<
    attendees.size() << " Total meals: ";
  std::cout <<
    meal_choices[0].get_total_meals();
  for(auto meal : meal_choices) {
    std::cout << " " << meal.meal_name << ": "
      << meal.get_count();
  }
  std::cout << "\n\n" << std::setw(9) <<
    "Attendees" << ":";
  std::string separator = " ";
  for(auto participant: attendees) {
    std::cout << separator <<
      participant.first;
    separator = ",  ";
  }
  for(auto meal: meal_choices) {
    std::cout << "\n\n" << std::setw(9) <<
      meal.meal_name << ":";
    int  id = meal.id;
    separator = " ";
    for(auto participant: attendees) {
      if(participant.second & id) {
        std::cout << separator <<
          participant.first;
        separator = ",  ";
      }
    }
  }
  std::cout << std::endl;

}
-- sample input --
  Roger breakfast lunch dinner
  John  lunch dinner
  Peter dinner
  Dave

  Paul  dinner lunch
  Mary  dinner
  Sadie high-tea dinner
  Peter dinner lunch
  Dave

  Kevin breakfast lunch lunch
  Huey  lunch
  Alvin
```

Felix Petriconi <felix@petriconi.net>

The main problem of the code is in the definition of the enum at the very beginning:

```
  enum class meal : int
  {
    breakfast,
    lunch,
    dinner
  };
```

Later in the code the enum is used as a bit field, so the values were combined with plus and or operators. But the enum values were not defined as a bitfield. Per default the first enum is initialised with zero and the next ones follow continuously. So the present code is equal to

```
  enum class meal : int
  {
    breakfast = 0,
    lunch = 1,
    dinner = 2
  };
```

So a combination of `breakfast | lunch` is equal to `0 | 1` which is equal to `1`. That means that the information of `breakfast` would be lost.

The bitfield should be corrected to

```
  enum class meal : int
  {
    breakfast = 1,
    lunch = 2,
    dinner = 4
  };
```

Further this should be improved:

The choice of enum `class` at the beginning is good. This C++11 feature ensures that potential accidental conversions to an integer type will not happen.

The loop inside the following stream operator **`std::istream &operator>>(std::istream &is, meal &m)`** should be changed to

```
for (const auto& p : names) {
  if (p.name == name)
    m = p.value;
}
```

because the existing code would create a copy of **`p`** for every loop iteration, which is a waste of resources. The same is valid for the loop inside the **`std::ostream &operator<<(std::ostream &os, meal const m)`** operator.

The operator

```
constexpr meal operator+(meal a, meal b) {
  return meal(int(a) + int(b));
}
```

tries to combine enumerated values but it may return values that are no valid meal. So a dinner + lunch results into a 2+4 (or 1+2 in the original code) None of the results is a valid meal value. So the operator should not return a type of meal, but an **`int`**.

The same is true for the **`meal operator+=(meal &a, meal b)`**.

So all operators would become:

```
constexpr int operator+(meal a, meal b) {
  return int(a) + int(b);
}
int operator+=(int &a, meal b) {
  a = a + int(b);
  return a;
}
constexpr int operator|(meal a, meal b) {
  return int(a) | int(b);
}
constexpr int operator&(int a, meal b) {
  return a & int(b);
}
```

The following **`static_assert`** is a great way to ensure that the bitfield uses each enum value exclusively. Since I changed the results type of the operators, the **`static_assert`** would have to be changed to

```
static_assert((int(meal::breakfast) |
  int(meal::lunch) | int(meal::dinner)) ==
            (int(meal::breakfast) +
  int(meal::lunch) + int(meal::dinner)),
  "not distinct");
```

The problem of possible invalid enum values implies that the struct

```
struct attendee
{
  std::string name;
  meal meals; // set of meals
};
```

should be changed to

```
struct attendee
{
  std::string name;
  int meals; // set of meals
};
```

The type definition **`using attendees = std::list<attendee>;`** should be changed to **`using attendees = std::vector<attendee>;`** because all non array like types have a very bad cache locality. Sometime ago I read the good advice, that every usage of a different container than array or vector should be explicitly justified.

I had to change the **`get_attendees`** function on my Mac to

```
attendees get_attendees(std::istream &is) {
  attendees result;
  std::string line;
  while (std::getline(is, line)) {
    std::istringstream iss(line);
    if (line.empty())             // new line
      return result;              // new line

    std::string name;
    iss >> name;
    meal meal;
    int meals{};
    while (iss >> meal)
      meals += meal; // add in each meal
    if (is.fail())
      throw std::runtime_error("Input error");
    result.push_back({name, meals});
  }
  return result;
}
```

Otherwise the **`while`** loop did not terminate.

Inside the **`while`** loop the variable **`meals`** is default initialised by using **`{}`** to zero which is in the old code **`breakfast`**, even if the attendee might not have breakfast. So I would extend the enum class with a **`none = 0`** enumerated value so that a **`{}`** results in an initialised variable with no value.

The function **`count`**

```
size_t count(attendees a, meal m) {
  size_t result{};
  for (auto &item : a) {
    // Check 'm' present in meals
    if ((item.meals & m) == m)
      ++result;
  }
  return result;
}
```

is written with the correct intention in mind. But it fails when the first enumerated value has the implicit value of zero. The expression of **`(time.meals & 0) == 0`** is always true. This results in the described failure of three breakfasts. Within this for loop I would change the type of the loop variable from **`auto&`** to **`const auto&`**, because the function has const semantics. As well there is no need to pass the **`attendees a`** parameter by value. In this case **`const &`** would be better, so the unnecessary copy of the complete container could be avoided. So the improved version would look like:

```
size_t count(const attendees& a, meal m) {
  size_t result{};
  for (const auto& item : a) {
    // Check 'm' present in meals
    if (static_cast<meal>(item.meals & m)
        == m)
      ++result;
  }
  return result;
}
```

In a next step I would change the routine by using an STL algorithm, because here a reduce is implemented by hand.

```
size_t count(const attendees& a, meal m) {
  return std::accumulate(a.cbegin(), a.cend(),
    0,
    [m](std::size_t r, const auto& item) {
       return r + (((item.meals & m) != 0)
         ? 1 : 0);
    });
}
```

The **`main`** function is a little bit unusual, because its body is the **`try–catch`** block. So I would slightly change it to:

```
int main() {
  try {
    auto attendees{get_attendees(std::cin)};
    std::cout << "Total: " << attendees.size()
      << '\n';
    for (auto m : {meal::breakfast,
                   meal::lunch, meal::dinner}) {
      std::cout << ' ' << m << ": " <<
        count(attendees, m);
    }
    std::cout << '\n';
    return 0;
  }
  catch (const std::exception &ex) {
    std::cout << ex.what() << '\n';
  }
  return -1;
}
```

So I recommend to add a newline at the end of the 'Total' line. From my point of view that would improve the readability of the output. Initialising the ranged based for loop per value is fine, because it is a loop over integral values and there is no performance penalty compared to a `const&` value.

At the end of the output I would return in case of an overall success a zero and in case of a failure a non-zero value. That is the common behaviour of programs.

James Holland <james.holland@babcockinternational.com>

The student has recognised that the meal enumerations have to be distinct and has had the foresight to construct a `static_assert` in an attempt to enforce that condition. Unfortunately, there are two problems associated with the student's code. Firstly, not only do the enumerations have to be unique, they have to contain one, and only one, bit position that has a value of '1'. This is so that each bit position within the enumeration is associated with a single meal type. The second problem is that the student's `static_assert` does not fully test for these conditions.

Making the meal enumerations fit the above requirements is simple; just make each a successive integer power of 2. This gives the enumerations `breakfast`, `lunch` and `dinner` the values of 1, 2 and 4 respectively. Enforcing this at compile-time is a little more difficult, however.

The approach I have adopted is to count the number of '1's in each enumeration and to ensure it is equal to one. I have written a `constexpr` function to do this. Then, to ensure each enumeration is unique, I 'or' all the enumerations and compare the result with the number of enumerations I am 'or'ing, in this case, three. Thanks to `constexpr`, all this is done at compile-time.

```
// Count the number of 1 bits.
constexpr size_t number_of_one_bits(
  const meal & m)
{
  using meal_type =
    typename std::underlying_type_t<meal>;
  size_t number_of_ones = 0;
  for (size_t bit_position = 0;
       bit_position <= 8 * sizeof (meal_type);
       ++bit_position)
  {
    if (static_cast<meal_type>(m) &
      (1L << bit_position))
    {
      ++number_of_ones;
    }
  }
  return number_of_ones;
}
static_assert(
  number_of_one_bits(meal::breakfast) == 1 &&
  number_of_one_bits(meal::lunch) == 1 &&
  number_of_one_bits(meal::dinner) == 1 &&
  number_of_one_bits(meal::breakfast |
    meal::lunch | meal::dinner) == 3,
  "Meal enumerations are not distinct.");
```

Finally, I am not sure how the student intended to signal to the program that all attendees' meal requirements have been entered. I have amended the software so that a blank line will bring things to a satisfactory close by employing `line.empty()` as shown below.

```
  while (std::getline(is, line), !line.empty())
```

Jason Spencer <contact@jasonspencer.org>

The basic problem with the code is that the `meal` enum is being used as a bit mask but actually expresses a bit position. Enum entries (scoped and unscoped) are assigned values which increment by one from the previous entry, starting from 0 for the first entry, unless the value is explicitly specified. So in the code `meal::breakfast` has a value of 0 (0000b), `meal::lunch` a value of 1 (0001b), and `meal::dinner` a value of 2 (0010b). I believe the intent of the programmer is to use the enum as a bitmask where every enum value has a different single bit set, so in this case the least significant bit (LSB) should signify breakfast, the second LSB lunch and the third LSB dinner – the values of which should be 0001b, 0010b, 0100b. The simple solution would be to make the enum:

```
  enum class meal : int { breakfast = 1,
    lunch = 2, dinner = 4, };
```

or

```
  enum class meal : int { breakfast = 1 << 0,
    lunch = 1 << 1, dinner = 1 << 2, };
```

My preferred solution would be to keep the enum the way it is, consider it a bit position rather than a bitmask, and wherever the meal bits are set or tested use `(1 << meal)` instead of just `meal`.

But I'll come back to the design later. There are other problems in the code, in order of the listings:

meal.h:

- `<iosfwd>` should be `<iostream>` – the stream operators are actually called on istreams and ostreams in the code, so we use more than just forward declarations. Yes, `sstream` will include `iostream` anyway, but it's more robust to include `iostream` directly if `sstream` is ever removed...
- `<sstream>` should be removed – no `stringstreams` are used in this file.
- Don't use a signed type for the enum underlying type – whether the enum is a bit position, or a bit mask. You lose one bit for the sign, and if you ever shift the bits you might run into trouble with sign extension. Consider perhaps using an unsigned integer type with a specific size e.g. `uint16_t`, `uint32_t` or `uint64_t` (from the `<cstdint>` header) – this way it'll be easier to read your intent with respect to memory usage. And if the mask is ever stored in another struct the issue of padding and storage can be more easily addressed.
- Consider renaming the `meal` enum to `meal_t` – so then there's no confusion between the meal type and the instance (see my `(1 << meal)` sentence above).
- `struct names[]` should be const as it's a lookup table which isn't expected to be changed. And if some code does change it, it may break the enum to string search loops.
- the `operator>>(istream,meal)` overload:
  - The range for loop doing a linear search should be `const auto & p : names` instead of `auto p : names` to prevent an unnecessary copy or an accidental overwrite of `p`.
  - Currently, if the meal isn't found in names then the word is still consumed from the stream, without an error being set, or the meal result variable being updated. The `failbit` must be set on the `istream` when the meal isn't found. This is done with `is.setstate(std::ios_base::failbit);` Note that `setstate` could also throw an exception at this point, depending on prior calls to `istream::exceptions(..)`.

    - Prefer using **`istream::sentry`** to trim whitespace from the input stream and to skip processing if the input stream already has the **`failbit`** set.

      You might not be expecting whitespace, but you don't know where the **`operator>>`** overload will be used in future, so don't rely on it being stripped (although of course **`is >> name`** should strip it for you anyway, so this isn't a hard rule).

- the **`operator<<(ostream,meal)`** overload: Nothing is printed if the meal isn't found in **`names`**. Also, if multiple bits are set then all recognised meals would be printed, but without a delimiter. If the bit position is not found in names consider whether an exception should be thrown, or the **`failbit`** set on the **`ostream`**, or whether you just print the raw bitmask (as a binary string)? The **`const`** ref is again missing from the range for loop search.
- **`operator+(meal,meal)`**, **`operator|(meal,meal)`** and **`operator&(meal,meal)`** would preferably have **`(const meal, const meal)`** arguments since we're not planning on changing them.
- **`operator+=(meal & a, meal b)`** should return an lvalue, so the return type should be **`meal &`** and not **`meal`**: **`(meals+=lunch)+=dinner`** should be valid. The second arg, **`meal b`** could be **`const meal b`** as we don't expect to change it.
- In the enum manipulation functions (that is the overloads of +, +=, |, &) – I'd strongly advise against hardcoding the int type into the operation – consider

  ```
  meal(static_cast<
    std::underlying_type<meal>::type>(a)+...)
  ```

  instead of **`meal(int(a)+...)`** – this way if you change the enum's underlying type your arithmetic won't break and there's much less maintenance. C++14 has the slightly shorter **`std::underlying_type_t<meal>`** over C++11's **`std::underlying_type<meal>::type`**.
- The **`static_assert`** test isn't perfect – it wouldn't catch the enum values set to { breakfast = 0, lunch = 0, dinner = 0, }. There's also a question over what distinct means – unique values or no overlapping set bits? This seems to test for no-overlapping bits, so adding a fourth meal **`{ breakfast, lunch, dinner, beer }`** (i.e. as a bit position enum, and not the shifted bit mask enum) would cause the assert to fail. There's also no equivalent test that **`::name`** in **`names[]`** are unique, although that might be overkill. Some might argue the test of uniqueness in the enum is overkill.

meals.cpp:

- **`#include <string>`** and **`#include <sstream>`** are missing. There's no compilation error because they were already included in `meals.h`, but if that ever changes (**`sstream`** should be dropped from meals.h anyway), it's best to include here.
- consider renaming **`attendee`** type to **`attendee_t`** to prevent confusion with an instance.
- **`attendee::name`** could be **`const`** as there's no obvious use case for changing the name after the struct is created.
- in **`get_attendees(..)`**:
    - Perhaps consider a rename to better express that meals are also read? Maybe **`read_attendees_meals`**? **`get`** usually implies an accessor, not a stream reader.
    - Regarding the initialisation of meal and meals in **`meal meal, meals{};`**: the **`meal`** variable is uninitialised, while **`meals`** is initialised to the default value, which *should* be 0. Perhaps consider initialising to 0 explicitly, since that's what we want in an empty mask. If **`iss >> meal`** cannot find the meal in names it won't update the meal variable – and then meals will either be corrupted by the uninitialised meal, or will have another meal added of the value of the previously read meal (which complicates things even more when you consider the next point).
    - The use of arithmetic add to add a meal in **`operator+=`** called by **`meals += meal;`** would mean that adding two lunches would become a single dinner (easy to do since we don't test for duplicates, or report errors). That's wrong.
- **`std::istream &operator>>(std::istream &is, meal &m)`** doesn't set the istream's fail bit if the meal isn't recognised, so **`get_attendees(std::istream &is)`** won't catch unknown meals. Worse still the **`meal`** variable in **`get_attendees`** is uninitialised and may now not be set in the read, while still being 'inserted' into meals.
    - There is no testing whether the read value for **`meal`** is already set in **`meals`**.
- In **`main()`**:
    - The main function is a special case when it comes to return values in that when there is no return statement an implicit **`return 0;`** is added (3.6.1.5 of [1]). In the case of a function-try-block when the function is **`main`** the compiler should do this at the end of the **`try`** clause.

      However, when reaching the end of the **`catch`** clause in the function-try-block the behaviour is equivalent to a return with no argument (15.3.14 of [1]), and therefore in this case the exit code is undefined. An explicit **`return 1;`** statement should be added to the end of the catch clause.
    - Rename **`count(..)`** to **`count_meals(..)`** – a function called 'count' could count anything, and could be confused with **`std::count`**. By the way – **`count_meals`** could easily be implemented with **`std::count_if`**:

```
size_t count_meals(const attendees_t a,
  const meal m) {
  return std::count_if( std::begin(a),
    std::end(a), [ m ] (
      const attendees_t::value_type & item ) {
       return (item.meals & m);
  });
}
```

    - when outputting the meals in the **`for`** range loop an **`initialiser_list`** of enums is used – this is yet another location to update if the enum list is ever changed. Unless a specific order of meals output is required consider using the enum values in **`names[]`**:

      ```
      for (const auto & mn : names )
        std::cout << ' ' << mn.name <<  ": "
          << count_meals ( attendees, mn.value );
      ```

In terms of general design:

- If new meals are ever added to the **`meal`** enum in `meal.h` there are four places that need to be updated – the enum, names, the **`static_assert`** and the list in **`main`**. Consider grouping at least the first three in the header file for easier updating.
- Consider changing the output text to say *Number of guests:* instead of *Total:* as the latter could be interpreted as the total number of meals.
- Consider moving the enum-to-string and string-to-enum mapping to two new functions – it'll make testing easier, the code will be self documenting, error handling can be made cleaner and the code may be reused elsewhere. These could be methods in **`attendee_t`**, although ideally the responsibility would be put in a new **`meal_t`** class alongside other helper functions and tests.
- in **`get_attendees(..)`** consider splitting the stream reading from the input parsing – in the future the reading and parsing can then change independently. Meals and attendees can also then be parsed from a string, without having to create a **`stringstream`** object.

- Although it’s syntactically sugary, I’d consider dropping the enum +, +=, |, & overloads – since we’re going to the trouble of naming the meals and having an enum for them, why confuse things by using cryptic symbols to manipulate them? `meals += meal` might be more intuitive than `meals |= meal`, but also possibly wrong. And is the `meal` enum type (meal, singular) a valid place to hold multiple meals (as the superposition of meals in a bitfield) ? The merged value is no longer one of the listed enum tokens, and is that conceptually still a meal?

  Do we even need both `operator|` and `operator+`? If it’s just used for the `static_assert` then do the addition there rather than making a user think it’s valid for general use – an API is as much about preventing users from doing things, as it is about facilitating things.

I see four points for further investigation:

Information encoding

Do we need a bitmask? Yes, it’s very memory efficient, but it has a limited and fixed number of available flags that are stored in it. If the meal names are hard-coded into an enum each bit field must have a given meaning at compile time. What if new meals are introduced, or the meals of multiple days need encoding in the future? a re-write and re-compile is needed (making sure to catch all four places that need the enum list needs to be updated).

My preferred design is to take a list of meals on the command line, and for each input line either put the meals as strings in an `std::unordered_set`, or an `std::unordered_multiset` (if in the future you want to allow multiple instances of a meal), without a bitmask. Or read the meal names as strings and directly increment a map like `std::unordered_map<std::string, size_t> meal_tally`;

Alternatively, encode the meals (the valid values of which are again specified on the command line) into a `std::bitset` (you’d need to specify a compile time limit on the number of meal options) or a `std::vector<bool>`. You have the meal names on the command line so can easily map bit positions to and from meal names. The ordering of meals on the command line can also specify the output ordering.

The drawback of dynamically specifying names is that if the bitmask is ever serialised as an integer the bit position meanings must be encoded separately.

Bitmask use

I would urge caution on using the enum type you’ve created to represent multiple meals as a bitmask. Their superposition has a different meaning to the original enums, and possibly has a conflicting absolute value. If you do want to use bitmasks, and you want the bit meaning to be specified at compile time as an enum, then use the enum as a bit position. Make it a scoped enum with an underlying type that is an unsigned integer. Have a typedef specifying an unsigned integer type which is called `mealset_t` (for the actual bitmask), or use an `std::bitset`, since the latter will deal with the bit positions and offsets for you. But don’t use the `meal` enum type as a bitmask type, because you’d be killing the meaning.

To get an idea of how an enum bitmask could be implemented have a look at 17.5.2.1.3 in [1].

enum management

C++ has very limited introspection, so you can’t at runtime convert an enum entry into a string with the human-readable name, and you can’t iterate over the enum values, nor even get the number of entries. Enums are just a bunch of scoped, or unscoped, constants. You could, however do things like:

```
enum class meal_t : uint32_t { breakfast,
  lunch, dinner, leftovers, NUM_MEALS };
const char * meal_names[] = { "breakfast",
  "lunch", "dinner", "leftovers" };
static_assert(
  std::size(meal_names)==NUM_MEALS);
```

which is similar to what the student did with `struct names[]`, but this suffers from poor maintenance, so you could get a little hacky with Boost.Preprocessor [2] and use the compiler pre-processor to automate the code generation:

```
#include <boost/preprocessor/seq/for_each.hpp>
#define STRINGIFYADDCOMMA_(s) #s,
#define STRINGIFY_ADD_COMMA(r, data, elem)\
  STRINGIFYADDCOMMA_(elem)
#define COMMAIFY(r, data, elem) elem,
#define PREFIX_COMMA(r, data, elem)\
  data::elem, using utype = uint32_t;

#define MEALS_SEQ\
  (breakfast)(lunch)(dinner)(beer)(icecream)
enum class meal_t : utype {
  BOOST_PP_SEQ_FOR_EACH( COMMAIFY, DUMMY,
MEALS_SEQ)
};
constexpr meal_t allmeals [] = {
  BOOST_PP_SEQ_FOR_EACH( PREFIX_COMMA,
  meal_t, MEALS_SEQ)
};
const char * meal_names[]= {
  BOOST_PP_SEQ_FOR_EACH( STRINGIFY_ADD_COMMA,
    DUMMY, MEALS_SEQ)
};
```

It’s somewhat of a mess, I know, and the code is a little difficult to read, but to iterate over all `meal` enums you just iterate over `allmeals`, to map names, use `meal_names[]`, to get number of meal types use `std::size(allmeals)`. And if you need to add new meals, you just add them to MEALS_SEQ and the rest is automatically generated. In GCC you an use the -E switch to see the output of the pre-processor.

Or you could just use a library like Better Enums [3].

iostream overloading

If you’ve overloaded a stream operator then there are a number of expectations. The expectations are in regard to exception handling and error reporting. As mentioned already the `istream`’s `failbit` must be set if formatted input is expected and it is a different format. Further, the streams and data must be left in a consistent state. Most formatted stream overloads leave, on bad input, the stream with the `failbit` set, no update of the variable, but possibly some characters taken from the input stream. This is a basic exception guarantee, and a strong guarantee would be hard or impossible to do with a stream (it may not be possible to roll back the underlying buffer or look-ahead far enough). All the details are covered in [4] and it is probably the go-to book for IOStreams – although the book is a little dated, the principles are all there.

It’s a good exercise in stream use, but instead of the overloads I’d recommend writing string to enum conversion functions and leave the I/O to the enum user.

In terms of further reading, apart from what is already mentioned above, [5] may be a useful reference to the state of streams (section 15.4) and exception use (section 15.4.4), as well as sentry objects (section 15.5.4) and `std::bitset` (section 12.5). In fact section 12.5.1 has an example use of an enum to specify bit position in a bitset used as a bitmask.

Of course, you could also implement the whole thing about two orders of magnitude faster in awk:

```
echo -e "Roger breakfast lunch\nJohn lunch
dinner\nPeter dinner" | awk
'{ for(i=2;i<=NF;++i) ++cnt[$i] } END { printf
"Guests: " NR; for (i in
cnt) printf " " i ": " cnt[i]; printf"\n"; }'
```

References

[1] *Working Draft, Standard for Programming Language C++*, n4296, 2014-11-19
[2] http://www.boost.org/doc/libs/release/libs/preprocessor/
[3] http://aantron.github.io/better-enums/

[4] *Standard C++ IOStreams and Locales: Advanced Programmer's Guide and Reference* by Angelika Langer and Klaus Kreft, Addison Wesley, 1st edition (31 Jan. 2000), ISBN 0321585585

[5] *The C++ Standard Library: A Tutorial and Reference 2nd Edition* by Nicolai M. Josuttis, Addison-Wesley, 25 May 2012 ISBN 0132977737

Commentary

The basic problem in the critique – that of using an enumeration to contain a set of values – is one that crops up in various guises. I would suggest that defining arithmetic operations on an enumeration is a category error.

While, as noted in some of the critiques, use of an enum is very efficient in terms of data storage it does have some other issues with usability. I suspect data storage is unlikely to be a serious problem for this program so I would probably recommend using an alternative design.

It might be worth exploring **`std::bitset`** as this already provides the logical operators, but the simplest design might just be to use a **`std::set`** of enumeration values (or even of strings) unless and until performance or memory use is a constraint.

Unfortunately, C++ does not provide any assistance with ensuring that enumerated values used as a bitmask are 'sane' (each defines a single bit with no overlaps). C# has an attribute (**`Flags`**) which at first sight looks useful, but it does not affect nor check the values in the enum.

Another problem with bitmasks is that of the detection of the 'no bits set' case. I recall when Microsoft Windows added a new value to the bitmask of file attributes, **`FILE_ATTRIBUTE_NORMAL`**, which is a simulated attribute that is set when no other attributes are set. The intent was that the 'no flags set' case could be detected by checking for this 'special' bit, but its existence caused a number of other problems and the need in some cases to special-case this value.

The Winner of CC 107

All the critiques that engaged with the code identified the fundamental problem with a bitmask value for **`breakfast`** of **`0`**. Jason's approach of keeping the implicit assignment of enum values and using bit shifting operations could a good way to proceed as it works with the flow of the language idioms; I'd probably want to see the bit shifting encapsulated to avoid confusion for the user.

While combining enumeration values with logical 'or' can produce values that are not in the list of named values this is not a problem, syntactically anyway, in C++ as the standard is careful to define the valid range of an enumeration to include these unnamed values. The danger of converting to the underlying type is that type safety is lost.

Jason's point about the number of places to change if a new meal type were added is a good teaching point to make to the author of the code. These sort of hidden dependencies in code bases make changes significantly harder, and learning to avoid creating them in the first place is good practice.

I have a great deal of sympathy with Russel's approach to the problem (echoed by Jim as well) that this problem does not seem to be a natural fit for the language used.

As Titus Winters pointed out in his keynote that this year's CppCon, one of the oddities of programming is that the lifetime of a program can vary by many orders of magnitude. *Efficiency* (whatever that means) must factor in the cost of writing and modifying the program; since the author here states they are writing a program for a one-off event the time spent developing the program is likely to significantly exceed the total time spent executing it. Ease of development then becomes very significant. So I'm not persuaded that using Boost.Preprocessor is a good solution to this particular problem as I suspect that might increase development time!

I was unable to explain Felix's problem with requiring a check on **`line.empty()`** – perhaps someone more experienced with the internals of the Mac C++ runtime can explain this? (The extra check does no harm however, and may even make the intent clearer.)

Several people mentioned the need to take care when overloading streaming operators to ensure the flags are set correctly, especially on input. In many cases this happens incidentally, as a call inside the implementation of the operator fails and sets the flags; this critique demonstrates one of the cases where a fully fledged solution needs to ensure the **`failbit`** is set manually on error.

Overall I am awarding Jason the prize for this critique as I thought he both covered the presenting problems well but also gave some other good suggestions for the author to improve their skill at programming.

Code Critique 108

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

> I've got a problem with an extra colon being produced in the output from a program. I've stripped down the files involved in the program a fair bit to this smaller example. Here is what the program produces:
>
> ```
> test_program Example "With space"
> 1:: 1001:Example
> 2:: "1002:With space"
> ```
>
> I can't see where the *two* colons after each initial number come from as I only ask for *one*.

Please can you help the programmer find the cause of their problem and suggest some other possible things to consider about their program.

- Listing 3 contains `record.h`
- Listing 4 contains `record.cpp`
- Listing 5 contains `escaped.h`
- Listing 6 contains `test_program.cpp`

Listing 3

```
namespace util
{
  class Record {
  public:
    Record(uint64_t id,
      std::string value = {});
    std::string to_string() const;
    // other methods elided
  private:
    uint64_t const id;
    std::string value;
  };
  inline
  std::string to_string(Record const &r)
  {
    return r.to_string();
  }
}
```

Listing 4

```
#include <cstdint>
#include <sstream>
#include <string>

#include "record.h"

util::Record::Record(uint64_t id, std::string
value)
  : id(id), value(value) {}

std::string util::Record::to_string() const
{
  std::ostringstream oss;
  oss << id << ":" << value;
  return oss.str();
}
```

Planet Code

Andy Balaam introduces his blog aggregator.

Since 2011 I have been maintaining a blog aggregator 'Planet Code' [1] that collects together blogs by people interested in coding. Many of the blogs that are aggregated are written by ACCU members, but it's not an official ACCU web site.

(By the way, I'd be happy to help set up an official one, or transform Planet Code into one, if the ACCU committee wanted to do that.)

If you'd like to suggest a site to add to Planet Code, including your own, or ask for a site to be removed (also including your own!) please drop me an email on andybalaam@artificialworlds.net.

I'd also welcome feedback on the site redesign I published this month, switching over to a WordPress platform instead of the static Venus system I was using before. Hopefully. this makes the site more mobile-friendly, readable, searchable and capable of aggregating feeds on modern HTTPS servers. Please let me know how it could be improved.

If anyone would like to help administer the site, making it a little more bus-proof, I'd like to hear from you.

If you're wondering how decisions are made about what to include, it's done in an informal way at the moment, with all decisions being made by me. If you'd like to raise any concerns with me, please get in touch.

I include feeds if:

- A significant proportion of their content covers programming topics, especially those I judge to be of interest to ACCU members. (This includes not too much blatant advertising!)
- They don't contain material that promotes hate or prejudice, or harasses or marginalises people and they don't contain imagery or language that would be unsuitable for children.

In most cases if I felt I needed to remove a feed I would contact its author to discuss it. In urgent cases, though, I might remove the feed before having that conversation. ■

Reference

[1] http://artificialworlds.net/planetcode/

ANDY BALAAM

Andy Balaam is happy as long as he has a programming language and a problem. He finds over time he has more and more of each. You can find his many open source projects at artificialworlds.net or contact him on andybalaam@artificialworlds.net

Code Critique Competition 108 (continued)

Listing 5

```
#pragma once
#include <string>

namespace util
{
  // provide 'escaped' textual representation
  // of value
  // - any double quotes need escaping with \
  // - wrap in double quotes if has any spaces
  template <typename T>
  std::string escaped_text(T t)
  {
    using namespace std;

    auto ret = to_string(t);
    for (std::size_t idx = 0;
      (idx = ret.find(idx, '"')) !=
        std::string::npos;
      idx += 2)
    {
      ret.insert(idx, "\\", 1);
    }
    if (ret.find(' ') != std::string::npos)
    {
      ret = '"' + ret + '"';
    }
    return ret;
  }
}
```

Listing 6

```
#include <cstdint>
#include <iostream>
#include "record.h"
#include "escaped.h"

using namespace util;

template <typename K, typename V>
void output(K key, V value)
{
  std::cout << escaped_text(key) << ": "
    << escaped_text(value) << '\n';
}

int main(int argc, char **argv)
{
  static uint64_t first_id{1000};
  for (int idx = 1; idx != argc; ++idx)
  {
    Record r{++first_id, argv[idx]};
    output(idx, r);
  }
}
```

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.

ACCU Information

Membership news and committee reports

accu

View from the Chair

Bob Schmidt
chair@accu.org

After last issue's detour, I'm back to my usual, chatty column. Let's get started…

International Standards Development Fund

At our September meeting, the committee voted to award an ISDF grant of £500.00 to Mr Walter Brown. Mr Brown was nominated for the grant by Roger Orr, for his contributions to the ISO C++ Standards effort. Mr Brown has recently retired, and it is hoped that the grant will assist his participation in future standards meetings. Mr Brown reports that he has six or so papers for consideration at the WG21 meeting in Albuquerque in November.

ACCU's ISDF is funded by contributions, which are kept separate from our operating accounts. Do you know someone who should be considered for an ISDF grant? Forward your nomination information to the committee – we will consider the nomination at our next scheduled committee meeting.

ACCU 2018

The next ACCU conference will be held in Bristol, UK, from the 11th through the 14th of April, 2018, with pre-conference workshops on April 10th. The conference again will be held at the Marriott City Centre, our home for the past several years.

Conference Chair Russel Winder reports that the proposal submission and review system is being rewritten. Russel says:

> If you fancy contributing to the usability and/or efficacy of this Flask/SQLAlchemy/Bootstrap3/Jinja2 system, head over to github[…] [1] and get stuck in. I have no budget for any paid work on this, but if there is anyone who makes outstanding contributions, I will certainly blag [2] them a free ticket to ACCU 2018.

Russel also reports that

> Austin Bingham has contributed his Elm-implemented ACCU schedule Web application. If anyone fancies working on that then head over to github[…] [3] and use the issues to debate ideas.

Web editing

Three of us on the committee have tried to take up some of the responsibilities of the web editor position. Meeting minutes are being posted. We've been successful at posting the PDF versions of the magazine, and have started getting caught up on posting the HTML versions. The by-author and by-article bibliographies are next on the list to tackle, followed by more regular updates of our social media channels. The e-pub versions of *C Vu* and *Overload* have been suspended until a new web editor can be found.

If you are interested in becoming web editor, please contact me at chair@accu.org.

History of ACCU

As I reported last issue, Matt Jones has been spearheading an effort to reconstruct the history of ACCU, concentrating on past committee members and honorary members. Matt reports that after an initial flurry of activity, incoming updates have started to slow down. Hopefully this means that we have fewer holes to fill in the history.

If you have information you would like to share, contact Matt at membership@accu.org.

Web site redesign

The committee discussed the idea of redesigning our web site using a newer content management platform. Currently the web site is implemented using Xaraya, which hasn't been updated in several years. We are soliciting ideas for a new platform, such as WordPress. If you have experience with a content management platform and would like to express your opinion on its '-ilities' [4], please send your comments to accu-committee@accu.com.

References

[1] https://github.com/ACCUConf/ACCUConf_Submission_Web_Application

[2] www.thefreedictionary.com **blag** – 1. to obtain by wheedling or cadging: she blagged free tickets from her mate.
My vocabulary is greatly expanded by reading Russel's emails.

[3] https://github.com/ACCUConf/ACCUConf_Schedule_Web_Application

[4] 'The 7 Software "-ilities" You Need To Know'
{codesqueeze}
http://codesqueeze.com/the-7-software-ilities-you-need-to-know/

Member news

Francis Glassborow contacted us with some exciting news:

I have just learnt that Bjarne Stroustrup has been awarded the 2017 Faraday Medal. It is the Institute of Engineering and Technology's highest award and previous recipients include such notable computer scientists as Maurice Wilkes, Professor Hoare, Roger Needham and, most recently, Donald Knuth."

Bjarne is receiving the award:

> For significant contributions to the history of computing, in particular pioneering the C++ programming language.

He is receiving it on 15th November.

“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

GET MOORE
intel Software
intel PARALLEL STUDIO XE
£634.99
TOOLS THAT EXTEND MOORE'S LAW
CREATE FASTER CODE—FASTER
Take your results to the next level with screaming-fast code.

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

To find out more about Intel products please contact us:

020 8733 7101 | enquiries@qbssoftware.com
www.qbssoftware.com/parallelstudio

qbs SOFTWARE