The magazine of the ACCU

www.accu.org

{cvu}

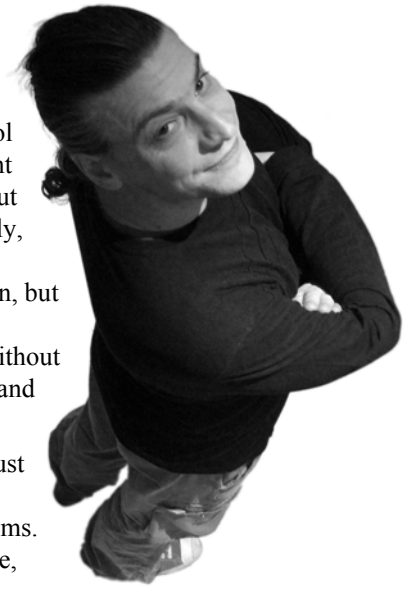
Volume 24 • Issue 6 • January 2013 • £3

Features

Regulars

{cvu}
Volume 24 Issue 6
January 2013
ISSN 1354-3164
www.accu.org

For The Sake Of It



It's a common enough thing to hear: "Change for the sake of change is counter-productive", or variations on the theme, like "If it ain't broke, don't fix it", and even "This is the way we've always done it".

It's certainly true that change can be a disruptive force, even a negative one. Deciding to implement your own tool instead of using an off-the-shelf and fully tested one might allow you better control over memory use or threading, but it might also introduce whole armies of bugs. Alternatively, adopting some shiny new technology or library into your codebase might save you hours of implementing your own, but might cost days or weeks in managing the additional unnecessary complexity. Making either kind of change without due consideration for the consequences will bring curses and derision from everyone.

Sometimes however, disruption can be a positive thing. Just mixing things up a bit can force people to think in new ways, and can present new perspectives on current problems. Being a force for *good* change requires vision and courage, even if the need for that change isn't always immediately evident. Someone, somewhere, once had the leap of imagination that led from rolling logs to axles. It wasn't that rolling logs were broken (alright, I bet they were from time to time...), rather that there was an alternative that just might be better, given a bit of a chance. The fact that the rolling-stuff-on-logs has been all but entirely replaced by wheels on axles is testament to the fact that the former was, in fact, broken.

Some ideas for change are entirely fashion – or cult – led, like the idea of mimicking the King's wearing of wigs: pointless but mostly harmless (wig makers probably enjoyed the idea very much!). Distinguishing among the genuinely visionary change, the latest fad, and the truly crackpot, can be tough, but that should not mean that we close our minds to *all* change, otherwise we'll never invent the new wheel.

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 25.1: 1st February 2013
C Vu 25.2: 1st April 2013

Overload 114: 1st March 2013
Overload 115: 1st May 2013

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

Impact of Semantic Association on Information Recall Performance (Part 1)

Derek Jones presents the analysis from his ACCU Conference experiment.

Comprehending source code involves reading it to obtain information which may only need to be remembered for a short period of time or may need to be remembered over a longer period.

One of the first major discoveries in experimental psychology was of a feature of human memory that has become commonly known as *short term memory*. People are able to temporarily retain a small amount of information in memory whose accuracy quickly degrades unless an effort is made to 'refresh' it, and the information is easily overwritten by new information.

This article reports on an experiment carried out during the 2012 ACCU conference that investigated the impact of a limited capacity short-term memory on subjects' performance in commonly occurring tasks that occur when comprehending short sequences of code. The tasks involved subjects' ability to recall some of the variable names appearing in the conditional expression of two `if`-statements and to correctly deduce which arm of an `if`-statement is executed.

The experiment is derived from and takes account of results from previous ACCU conference experiments. In particular the 2004 experiment [1], which also asked subjects to select which arm of an `if`-statement they thought would be executed, and the 2007 experiment [2], which found that developers appeared to make use of identifier names to make operator precedence decisions. It is hoped that this study will provide information on the effect that identifier names have on developer performance during program comprehension.

Information in short term memory is either quickly lost or stored in another, longer term, memory subsystem

Few developers appreciate how short the *short* in short term memory actually is; its capacity has been found to correspond to approximately two seconds worth of sound, with some people having less capacity and some more. This is sufficient to hold information on a few statements at most. The analysis of the results from this study will hopefully highlight to developers the consequences of short term memory limitations on their code comprehension performance.

This article is split into two parts, the first (this one) provides general background on the study and discusses the results of the `if`-statement memory/recall problem, while part two discusses selecting executed arm of `if`-statement problem.

The hypotheses

Measurements of source code [2] have found that some English words usually appear in variables that are the operands of bitwise or logical operators (e.g., flag) while some English words usually appear in variables that are the operands of arithmetic operators (e.g., count). The 2007 ACCU experiment found that developers appeared to make use of the occurrence of these words when asked to make binary operator precedence decisions.

The hypotheses tested by this experiment are that:

1. developers are less likely to confuse identifiers appearing as operands in an expression if the operand names are consistent with common usage for the corresponding binary operator,
2. developers are less likely to be able to correctly recall the identifiers appearing as operands in an expression (e.g., not remember their name or use a different name) when one or more identifiers are not consistent with common usage for the corresponding binary operator.

It is difficult to evaluate whether nonconsistent operand names will be less likely to be recalled because they fail to match a known pattern or be more likely to be recalled because they stand out as being different than expected (which may result in a reduction in resources used to remember consistent names leading to a greater chance of confusion among consistent names).

Characteristics of human memory

Models of human memory often divide it into two basic systems, short term memory (the term *working memory* is technically more correct, while short term memory can be applied to any one of several memory subsystems and long term memory). This two subsystem model is something of an idealization in that there is not a sharp boundary between short and long term memory; there is a gradual transition between them.

The *short term memory* subsystems are a gateway through which all input memory data input must pass. They have a very limited capacity and because new information is constantly streaming through them, a particular piece of information rarely remains in it for very long. Information in short term memory is either quickly lost or stored in another, longer term, memory subsystem.

Declarative memory is a long-term memory that has a huge capacity (its bounds are not yet known) and holds information on facts and events. Two components of declarative memory of interest to the discussion here are episodic and semantic memory. Episodic memory [3] is a past-oriented memory system concerned with *remembering*, while semantic memory is a present-oriented memory system concerned with *knowing*.

While some of the characteristics of human memory (e.g., forgetting) are often criticized, they do provide useful functionality. People who can readily remember and later accurately recall information report that their conscious thoughts are repeatedly interrupted by *unforgotten* information [4]. It would make sense for human memory to be optimized for the information recall demands that most commonly occur in everyday life and various studies [5] appear to confirm this evolutionary priority; there seems to be an exponential decay in the likelihood that information will be needed again after it is first encountered and this is the rate at which information is *lost* from memory. Thus the likely need for information and the rate at which it is forgotten decrease in the same way.

Studies have found that practising the remembering and recalling of information does not lead to a general improved performance on

DEREK JONES

Derek used to write compilers that translated what people wrote. These days he analyses code to try to work out what they intended to write. Derek can be contacted at derek@knosof.co.uk

Figure 1

```
---------first side of sheet starts here----------

if ( (cars + lorries) == amount )
   x = 1;
if ( (settings | register) == final )
   x = 0;



---------second side of sheet starts here---------

if ((e > a) && (u < a))
   if (u > e)

      ..............
   else
      ..............
if ( (_____________  ___  _____________) == amount )
   x = 1;
if ( (_____________  ___  _____________) == final )
   x = 0;
```

remember/recall tasks (i.e., in the sense that exercise can lead to muscle growth, leading to increased strength). However, extended practice does provide people with the opportunity to try out different information memorization strategies and there are cases where people have discovered strategies (e.g., using mnemonics having associations with a person's existing network of memories) that lead to significant performance improvements for particular kinds of information.

Experimental setup

The experiment was run by your author during a 40-minute lunch time session at the 2012 ACCU conference (www.accu.org) held in Oxford, UK. Approximately 370 people attended the conference, 22 (5.9%) of whom took part in the experiment. Subjects were given a brief introduction to the experiment, during which they filled in background information about themselves, and then spent 20 minutes answering problems. All subjects volunteered their time and were anonymous.

The problem to be solved

Each problem seen by subjects was intended to involve memory processes that operate over a time frame of approximately 30 seconds. It was expected that the characteristics of short term memory would have a significant impact on subjects performance within this time frame.

To obtain statistically reliable results a large number of answers to related problems are needed. Therefore it is necessary to create lots of variations to the same underlying problem, also problems should not be too easy or too difficult (if all subjects answered all question correctly, or all incorrectly, no useful information would be obtained). The intent is that the study thus has some claim to being *ecologically valid* (i.e., the behaviour in the experimental situation is characteristic of a real life environment).

Figure 1 is an example of one of the problems seen by subjects. One side of a sheet of paper contained three assignment statements while the second side of the same sheet contained the `if` statements and a table to hold the recalled information. A series of X's were written on the second side to ensure that subjects could not see through to identifiers and values appearing on the other side of the sheet. Each subject received a stapled set of sheets containing the instructions and 38 problems (one per sheet of paper).

In practice software developers do not make a remember/not remember decision, there is always the opportunity to refer back to previously read information. The selection remember/*would refer back* more accurately reflects the decision made by developers.

The instructions given to subjects followed that commonly used in memory related experiments. Subjects see the material to be remembered, then perform an unrelated task (chosen to last long enough for the contents of short term memory to have degraded), and then have to recall the previously seen information. The sequence 'remember→unrelated task→recall' has an obvious parallel in source code comprehension; i.e., 'sequence of assignments→conditional test→use of identifiers previously assigned to'. The written instructions given to subjects are shown opposite.

Generating the problem

Each problem contained two parts, the 'to be remembered information' question and the 'which arm will be executed' question. The generation of the 'to be remembered information' is covered here and generation of the other question is covered in the second part of the article.

The 'to be remembered information' was contained within an `if`-statement having the following form:

```
if ( (cars + lorries) == amount )
   x = 1;
```

with the expression to the left of the equality operation (either a `==` or `!=`) always containing a single binary operator and its two operands, and the expression to the right always containing a single identifier.

The binary operator was either arithmetic (one of `-` or `*`) or bitwise (one of `|` or `&`).

Each operand is an identifier consisting of a word consistent with or not-consistent with the operator, where consistency is defined as appearing in an identifier that commonly appeared as an operand of one kind of operator and rarely appearing as the operand of the other kind of operator.

In the following discussion *C* denotes an operand whose name is consistent with the associated operator and *N* an operand whose name is not consistent; *a* is an arithmetic operator and *b* is a bitwise operator. So *CaC* is an expression containing an arithmetic operator and two consistently named operands and $\frac{CaC}{NbC}$ is a pair of expressions.

The 'to be remembered information' consisted of a pair of expressions, each containing two identifiers and an operator. It was thought likely that

Instructions

This is not a race and there are no prizes for providing answers to all questions. Please work at a rate you might go at while reading source code.

The task consists of remembering the names of four different variables, plus two binary operators, and later recalling and writing them down. The variable names and operators are contained in two if-statements appearing on one side of a sheet of paper and your response needs to be given on the other side of the same sheet of paper.

1. Read the conditional expression contained in each `if`-statement as you would when carefully reading code in a function definition at work.
2. Turn the sheet of paper over. Please do **NOT** look at the `if`-statements you have just seen again, i.e., once a page has been turned it stays turned.
3. For the nested `if`-statement appearing at the top of the new page, please place a cross in those arms of the inner `if`-statement that you think could be executed. This could be only one arm or could be both arms of the `if`-statement.
4. You are now asked to recall some of the variables and operators seen on the previous page.
   - if you can remember the name of any variable write it in the corresponding underlined portion of the conditional expression.
   - if for one or more variable names or operators, you feel that in a real life code reading situation you would refer back to the previously seen `if`-statement, write 'refer back' for the corresponding variable name or operator.

If you do complete all the questions do **NOT** go back and correct any of your previous answers.

subjects would sometimes give answers where the order of operands within an expression would be swapped and that swapping might also occur for operands between expressions within a pair. The generation of the expression pairs was balanced so that any operand swapping could be analysed.

If recall performance is better when operand names are consistent and worse when operand names are not consistent (compared to neutral operand names), then between expression operand swapping is less likely to occur in the pair $\frac{CaC}{CbC}$ than in the pair $\frac{CaC}{CaC}$; the pair $\frac{CaN}{CbN}$ would be expected to be the most likely to produce answers containing a swap between expressions.

The following are the operand combination patterns generated for this experiment:

- The four possible operand/operator combinations where all of the operand names are consistent with their corresponding operator, i.e., $\frac{CaC}{CbC}$, $\frac{CaC}{CaC}$, $\frac{CbC}{CbC}$ and $\frac{CbC}{CaC}$
- the 16 possible operand/operator combinations where one of the operand names is not consistent with their corresponding operator, i.e., $\frac{NaC}{CbC}$, $\frac{CaN}{CbC}$, $\frac{CaC}{NbC}$ and $\frac{CaC}{CbN}$, and so on for the three other possible operator combinations,
- 24 of the possible operand/operator combinations where two of the operand names are not consistent with their corresponding operator. The 24 combinations contained three sets, eight where the two nonconsistent operand names were horizontally aligned (i.e., both appeared in the same expression, e.g., $\frac{NaN}{CbC}$), eight where the two nonconsistent names were vertically aligned (i.e., one in each expression and both in the same operand position, e.g., $\frac{NaC}{NbC}$) and eight where the two nonconsistent names were diagonally aligned (i.e., one in each expression and their appeared in different operand positions, e.g., $\frac{NaC}{CbN}$).

Table 1

The identifier names commonly occurring in the operands of the corresponding binary operator; anonymous names commonly occurring in operands to all binary operators.

Bitwise	Arithmetic	Anonymous
flag	offset	value
state	rate	field
bits	size	temp
options	length	i
status	count	data
mask	maximum	current
active	minimum	id
done	width	buf
started	index	
shift	height	
success		
reset		
0x17	25	

It was decided to concentrate the analysis on the above 48 combinations and not to investigate operand/operator combinations where three of the operand names are not consistent with their corresponding operator.

Measurements of source code from a previous experiment [2] were used to obtain three lists of English words, one containing words that primarily occurred in variables appearing as the operands of bitwise or logical operators, one containing words primarily occurred in variables appearing as the operands of arithmetic operands and the third containing words or letter sequences commonly occurring with all kinds of operators. The literals `0x17` and `25` were included so that the generated expressions were more representative of real code (which includes lots of literals). See Table 1.

The problems and associated page layout were automatically generated using a shell script and various awk programs to generate troff, which in turn generated postscript and this was printed to paper.

The operand combination pattern was randomly selected (the ratio 30% all operands consistent, 40% one operand not consistent, 30% two operands not consistent was used) and identifiers/constant from the required set were randomly chosen, once used the identifier/constant was not used again until all other identifiers in that set had been used. The order of the two statements (for each problem) was also randomized. The source code is available on the experiments web page [6].

Threats to validity

Experience shows that software developers are continually on the look out for ways to reduce the effort needed to solve the problems they are faced with. Because each of the problems seen by subjects in this study has the same structure it is possible that subjects will detect what they believe to be a pattern in the problems and then attempt to use this perceived information to improve their performance.

The context in which an expression occurs in real life code reading situations may provide additional semantic assistance in recalling identifier names and this assistance is not present in this experimental setting. One subject wrote about the lack of application domain context on the Comments page.

Several subjects commented that they started out with no memorization strategies and then adopted one after answering several questions. It is possible that this change of strategy may have had an effect on the distribution of the kinds of answers they gave.

While the kind of problems used commonly occur during program comprehension, the mode of working (i.e., paper and pencil) does not. Source code is invariably read within an editor and viewing is controlled via a keyboard or mouse. Referring back to previously seen information (e.g., expressions in statements) requires pressing keys (or using a mouse). Having located the sought information more hand movements (i.e., key pressing or mouse movements) are needed to return to the original context. In this study subjects only needed to leave the answer blank, or write '?'or 'refer back' to indicate that they *would refer back* to locate the information. The cognitive effort needed for these actions is likely to be less than would be needed to actually refer back. Studies have found [7] that subjects make cost/benefit decisions when deciding whether to use the existing contents of memory (which may be unreliable) or to invest effort in relocating information in the physical world. It is possible that in some cases subjects ticked the *would refer back* option when in a real life situation they would have used the contents of their memory rather than expending the effort to actually refer back.

Subjects were asked to specify *would refer back* as the answer if they felt that in a real life code reading situation they would refer back to the previously seen if-statement. Some subjects wrote "?" for the operator or operand name, while others wrote nothing. All three cases, "refer back", "?" and blank were treated as being equivalent.

Two of the words used (i.e., `bits` and `options`) were the plural form of the word. In a few cases answers failed to include the final letter 's' and sometimes the word `flag` with a final letter 's' was given as an answer. An end of word 's' was added/removed from an answer if this resulted in it becoming a correct answer (around 10 instances). In some cases it was not clear from the written answer what the final letters of some words were and they were assumed to be those needed to complete the corresponding word appearing in the question.

One subject wrote in the comments that they were not a native English speaker another that they were dyslexic.

One subject wrote in the comments that they answered the recall problem before answering the nested `if` statement problem. This subject had the 4th highest percentage of correct answers (see subject 19 in Figure 1; subject 1 in the raw data).

Results

It was hoped that at least 30 people (on the day 22) would volunteer to take part in the experiment and it was estimated that each subject would be able to answer 25 problems (on the day 19.6, sd 8.1) in 20-30 minutes (on the day 20 minutes). Based on these estimates the experiment would produce 750 (on the day 430) pairs of if statements were remembered/recalled.

The raw results for each subject and the scripts used to process them are available on the experiment's web page [6].

Expected direction of behaviours

- When all operand names are consistent with its operator subjects will be less likely to make between expression mistakes for $\frac{CaC}{CbC}$ compared to $\frac{CaC}{CaC}$ because any swapped operands would not be consistent with their operators; in the second form any swapping of operands is consistent with the operators,
- fewer correct answers when an `if` statement pair contains one or more operand names that are not consistent with their operator,

There is no expectation of behaviour bias for answers involving operator recall.

Actual results

Most of the results presented here take the form of 2-dimensional contingency tables (see Table 2). Given the hypothesis that the rows of the table are drawn from the same distribution (i.e., the same underlying process) a Pearson chi-squared test can be used to calculate the p-value for this hypothesis. A very low p-value (e.g., 0.01 or 1 in 100) is taken to imply that the rows are drawn from different distributions (i.e., there is a difference in the processes involved).

A low p-value tells us that randomly generated rows are unlikely to be this different unless they are generated by different processes.

How big an effect is the difference between rows? Cramer's V is used as a measure of effect size, its value varies between 0 (no effect) to 1 (complete effect).

An awk script was used to convert the answers to a comma separated list which was then processed using the R statistical package. The R function `assocstats` from the package `vcd` was used to obtain the values for the Pearson chi-squared test and Cramer's V.

Incorrect answers for the operand name are broken down into those that are wrong and those that are the name of a different operand appearing in the question (i.e., one of three possible other names). Writing an expression containing an operand name that is wrong might result in a compile time error while writing code an expression with operand names swapped is much less likely to generate a diagnostic from the compiler.

Table 2

Number of operand answers that are wrong, in various operand positions or subject *would refer back*.

	Wrong	1	2	3	4	Would refer back
1st operand	9	372	7	7	0	36
2nd operand	16	7	350	1	8	49
3rd operand	18	7	1	320	15	70
4th operand	5	1	7	10	331	77

Table 2 summarizes the operand answers. The rows are the operand numbers (first operand of first expression is number 1, second operand is 2, first operand of second expression is number 3 and its second operand is 4). The first column is the total number of wrong answers for each operand position The next four columns give the total number of answers appearing in different operand positions, for instance if the first operand in the first expression is `flag` and the answer gives this name as the second operand of the first expression then column 2/row 1 is incremented by one. The last column is the total number of *would refer back* answers for that operand.

All operand names consistent Table 3 contains the total number of answers in various categories for questions where all operands have names that are consistent with their corresponding operator; if an operand name appearing in the question appeared in the answer in an incorrect position it was counted as a *swapped* operand (column 4 in the table).

Table 3

Total, for all subjects (percentage for each row in brackets), of wrong, correct, *would refer back* and swapped operand answers for the four combinations of operator ordering where all operand names are consistent with their corresponding operator.

Operator pairs	Wrong (%)	Correct (%)	Would refer back (%)	Swapped (%)
$\frac{a}{a}$	4 (3)	99 (82)	13 (10)	4 (3)
$\frac{b}{b}$	5 (6)	103 (83)	10 (8)	6 (4)
$\frac{b}{a}$	2 (1)	113 (76)	29 (19)	4 (2)
$\frac{a}{b}$	3 (2)	115 (77)	25 (16)	5 (3)

To test hypothesis 1 the $\frac{CaC}{CaC}$ and $\frac{CbC}{CbC}$ totals are added together and also the totals from $\frac{CaC}{CbC}$ and $\frac{CbC}{CaC}$ are added, giving two rows. A Pearson chi-squared test on these two rows produces p-value=0.016, i.e., the rows are likely to be drawn from different distributions (1 in 62 chance of this result occurring if they are drawn from the same distribution).

Hypothesis 1 predicts that the row totals are drawn from different distributions, but predicts a direction of difference that is the opposite of that seen in the answers; in practice there were more *would refer back* answers when the operators are different (the hypothesis predicts less).

The value of Cramer's V is 0.138, a very small effect (while *would refer back* answers almost doubled the overall contribution of that particular kind of answer is much smaller than Correct answers).

Operand names not consistent Table 4 contains the total number of answers of various kinds for questions where all operand names are consistent, one operand has a name that is not consistent with its corresponding operator and two operands have names that are not consistent. The row percentages are remarkable similar and a Pearson chi-squared test gives p-values that reflect this fact; the consistency of operand/operator naming does not have any impact on subject recall performance (i.e., no support for hypothesis 2).

Table 4

Total, for all subjects, of wrong, correct, *would refer back* and swapped answers for the 8 cases where all operand names are consistent, the 16 cases where one operand has a name that is not consistent with its corresponding operator and 24 of the cases where two operands have names that are not consistent.

Operand kind	Wrong (%)	Correct (%)	Would refer back (%)	Swapped (%)
All consistent	14 (2)	430 (79)	77 (14)	19 (3)
One not consistent	16 (2)	552 (79)	87 (12)	37 (5)
Two not consistent	20 (4)	387 (79)	70 (14)	11 (2)

Impact of operand position

What impact does the relative position of the operand have on recall performance (i.e., first or second in the expression, or appearing in the first or second expression)?

Table 5

Total, for all subjects, of wrong, correct, *would refer back* and swapped answers all questions, first row the first operand of an expression and second row the second operand of an expression.

Operand kind	Wrong (%)	Correct (%)	Would refer back (%)	Swapped (%)
1st operand	27 (3)	692 (80)	106 (12)	37 (4)
2nd operand	21 (2)	681 (79)	126 (14)	34 (3)

Table 5 breaks down the total answer count by operand position, either to the left of the operator (first row) or the right (second row). A Pearson chi-squared test gives p-value=0.44, which strongly suggests that the row values are drawn from the same distribution (i.e., any difference in totals is likely to be the result of random variation).

Table 6

Total, for all subjects, of wrong, correct, *would refer back* and swapped answers all questions, first row the first expression and second row the second expression.

Operand kind	Wrong (%)	Correct (%)	Would refer back (%)	Swapped (%)
1st expression	25 (2)	722 (83)	85 (9)	30 (3)
2nd expression	23 (2)	651 (75)	147 (17)	41 (4)

Table 6 breaks down the total answer count by expression position, either to the first expression (first row) or the second expression (second row). A Pearson chi-squared test gives p-value=6.4e-05, which very strongly suggests that the row values are drawn from different distribution.

The value of Cramer's V is 0.113, a small effect (for the same reasons given above).

Does operator recall performance also vary between expressions?

Table 7

Total, for all subjects, of wrong, correct, *would refer back* and swapped operator answers for the first and second expression.

	Wrong (%)	Correct (%)	Would refer back (%)	Swapped (%)
1st operator	9 (2)	378 (87)	32 (7)	12 (2)
2nd operator	16 (3)	347 (80)	55 (12)	13 (3)

Table 7 summarizes the operator answers. A Pearson chi-squared test gives p-value=0.024 (1 in 42 chance of being drawn from the same distribution). There is a greater chance of a *would refer back* answer being given for an operator in the second expression, compared to the first.

Individual subject performance

Table 8 lists the mean and standard deviation of subject operand recall performance.

Table 8

Mean and standard deviation of the various kinds of subject answers (normalised as a percentage), i.e., mean and sd of values in Figure 2, plus the mean number of questions answered by subjects.

	Wrong	Correct	Would refer back	Swapped	Questions answered
mean	3.3%	75.8%	16.3%	4.5%	19.6
sd	3.1	19.4	15.3	4.0	8.1

Figure 2 is a plot of the total number of operand answers for each subject as a percentage of all answers they gave, the subjects are ordered in increasing percentage of correct answers. There appears to be a negative correlation between percentage of correct answers and *would refer back* answers, the Pearson correlation coefficient is -0.96 (95% confidence interval: -0.98 -0.91, p-value=8.6e-13). The only other strong correlation is between percentage of correct answers and swapped operand names, -0.76 (95% confidence interval: -0.89 -0.50, p-value = 4.1e-05).

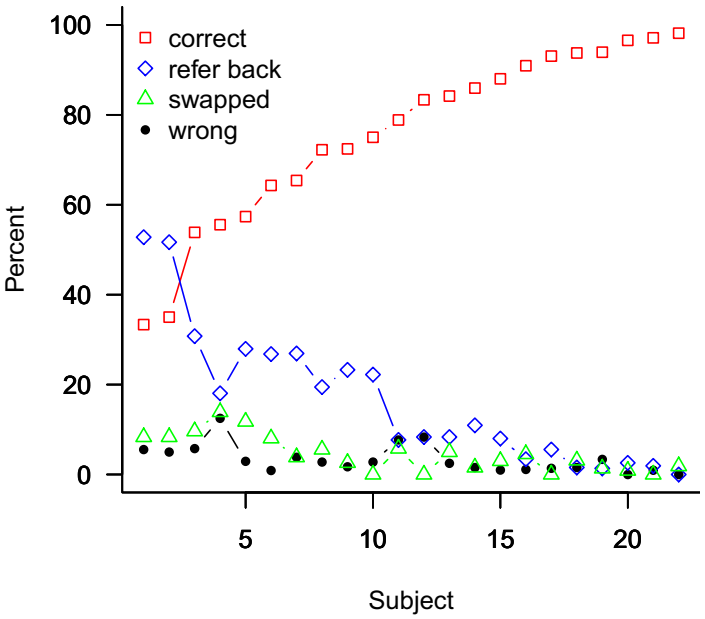
As in previous memory experiments [8] there were a few subjects who give a very high percentage of *would refer back* answers.

After correct answers *would refer back* is almost always the next most common answer, followed by operand names being swapped with another operand and then wrong answers.

The wide variation in individual subject performance suggests that the experimental design aim of creating a problem whose solution stretched the limits of subjects' short term memory capacity was achieved. Had the problems required more or less short term memory capacity then it is likely that the variations in subjects performance would have been smaller (i.e., subjects would have been likely to have given fewer or a greater number of wrong or *would refer back* answers).

Figure 2

The percentage of correct (square box), *would refer back* (diamond), swapped (triangle) and incorrect operand answers (bullet) for each subject. Subjects are ordered by percentage of correct answers given.



Discussion

The number of questions answered by subjects (19.6) is similar to the number of questions answered in 2004 (22.7) when the same nested conditional if statement was used between the remember/recall problem.

The largest effect found is that subjects' correct recall performance is lower on the second expression, compared to the first.

Experiments have shown that when subjects are asked to remember a list of words there is a *primacy effect* (i.e., they have better recall performance for items at the start of a list) and a *recency effect* (i.e., they have better recall performance for items at the end of a list) [9].

Is there a serial order in the if statement problem? With only two operands and two expressions every operand could be said to be either at the start or end of a list. Also the order in which subjects read and possible reread the information to be remembered is not known.

The order in which subjects wrote their answers is likely to be first expression followed by second expression. Perhaps the time taken writing the answer to the first expression caused information on the second expression to be lost from memory. Depending on the real life activity being performed this delay may or may not occur (i.e., a delay occurs when writing code but little delay need occur when a developer is processing code in their head)

In some contexts swapping an operand name is a mistake that could result in a fault being introduced into the code (e.g., if the operator involved was subtract or divide) and in other contexts swapping the operands of an operator is unlikely to result in a fault (e.g., the addition operator).

Conclusion

Subject operand recall performance, for a binary operator and the names of its two operands appearing in the control expressions of two consecutive if statements, was found to depend on:

- Whether the subexpression containing the operands appears in the first or second if statement (more *would refer back* answers are given for the second; not proposed as a hypotheses),
- whether the two if statement subexpressions, when they both have all operand names consistent with their respective operator, contains

the same or different operators (more *would refer back* answers are given when the operators are different; the opposite behavior predicted by hypothesis 1).

No statistically significant difference in operand recall performance was found between all operand names being consistently named, one operand not consistently named or two operands not consistently named (no evidence for hypothesis 2). ■

Further reading

For a readable introduction to human memory see *Essentials of Human Memory* by Alan D. Baddeley. A more advanced introduction is given in *Learning and Memory* by John R. Anderson. An excellent introduction to many of the cognitive issues that software developers encounter is given in *Thinking, Problem Solving, Cognition* by Richard E. Mayer.

Acknowledgments

The author wishes to thank everybody who volunteered their time to take part in the experiment, the ACCU for making a conference slot available in which to run it and Roger Orr for encouraging conference attendees to take part.

Also, advantage was taken of the `if` statements used in the experiment to try and duplicate the pattern of subject performance seen in some studies of human reasoning. The results seen in some of these studies suggest that the ordering of operands in a pair of relational expressions has an impact on people's performance in evaluating it.

References

[1] D. M. Jones, 2004, 'Experimental data and scripts for short sequence of assignment statements study.' http://www.knosof.co.uk/cbook/accu04.html

[2] D. M. Jones, 2008, 'Operand names influence operator precedence decisions' *C Vu*, 20-1 pp 5–11, Feb. 2008.

[3] A. Baddeley, M. Conway, and J. Aggleton, 2002, *Episodic Memory: New Directions in Research*, Oxford University Press.

[4] A. R. Luria, 1986, *The mind of a mnemonist*, Harvard University Press.

[5] J. R. Anderson and R. Milson, 1989, 'Human memory: An adaptive perspective', *Psychological Review*, 96(4) pp 703–719.

[6] D. M. Jones, 2012, 'Experimental data and scripts for impact of semantic association on information recall performance', http://www.knosof.co.uk/dev-experiment/accu12.html

[7] W. T. Fu and W. D. Gray, 2000, 'Memory versus perceptual-motor tradeoffs in a blocks world task' in *Proceedings of the Twenty-second Annual Conference of the Cognitive Science Society*, pp 154–159, Hillsdale, NJ. Erlbaum.

[8] D. M. Jones, 2012, 'Effects of risk attitude on recall of assignment statements', *C Vu* 23-6 pp19--22, Jan 2012.

[9] R. N. A. Henson, 1996, 'Short-term Memory for Serial Order', PhD thesis, University of Cambridge, Nov 1996.

Navigating a Route

Pete Goodliffe helps us to work on a new codebase.

The Investigation of difficult Things by the Method of Analysis, ought ever to precede the Method of Composition.

~ Sir Isaac Newton

A new recruit joined my development team this month. Our project, whilst not vast, is relatively large and contains a number of different areas. How could he plot a route into the code? From a standing start, how could he rapidly become productive?

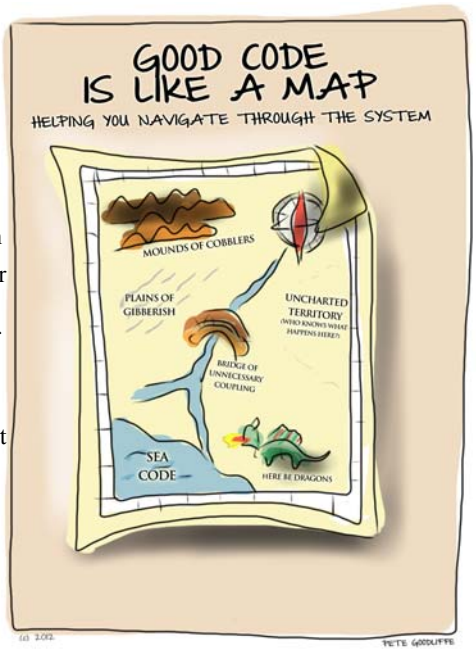
It's a common situation; one which we all face from time to time. If you don't, then you need to see more code and move onto new projects more often! It's important not to get stale from working on one codebase with one team forever.

Coming into any large existing code base is hard. You have to rapidly:

- Discover where to start looking into the code
- Work out what the code does, and how it achieves it
- Gauge the quality of the code
- Work out how to navigate around the system
- Understand the coding idioms, so your changes will fit in well
- Know where to look for the likely home of any functionality (and consequent bugs in that code)

You need to learn this quickly, as you don't want your first changes to be too embarrassing (in the best case), accidentally duplicate existing work, or break something elsewhere (in the worst case)!

In this article, we'll work out some practical ways to do this.



A little help from my friends

My new colleague had a wonderful head start in this learning process. He joined an office with people who already knew the code, who could answer innumerable small questions about it, and point out where existing functionality could be found. This kind of help is invaluable.

If you are able to work alongside someone already versed in the code then do so. Don't be afraid to ask questions. And if you can, take opportunities to pair program and/or get your changes reviewed.

> Your best route into code is to be led by someone who already knows the terrain. Don't be afraid to ask for help!

If you can't mither people nearby, don't fear; there may still be helpful people further afield. Look for online forums or mailing lists that contain helpful information and helpful people. There is often a healthy community that grows around popular open source projects.

The trick when asking for help is to always be polite, and to be grateful. Ask sensible, appropriate questions. "Can you do my homework for me?" is never going to get a good response. Always be prepared to help others out with information in return.

Of course, it's common sense, but do make sure that you've Googled for an answer to your question first. It's simply politeness to not ask silly questions that you could easily have found answers to yourself. You won't endear yourself to anyone if you continually ask basic questions and waste people's precious time. Like the boy who cried wolf and failed to get help when he really needed it, a series of mind-numbingly dumb questions will make you less likely to receive the complex help when you need it.

Look for clues

If you are rooting around in the murky depths of a software system without a personal guide to hand, then you need to look for the clues that will guide you around the code.

Some good indicators are:

- How easy is it to obtain the source and build it? This is a good indicator about the health and maturity of a project. Does it require installation of new tools? (How up-to-date are those tools?) Does one simple, single step build the entire system, or does it require many individual build steps? Does the build process require manual intervention? Can you work on a small part of the code, and only build that section, or must you rebuild the whole project repeatedly to work on a small component?
- How easy is it to build the code from scratch? Is there adequate and simple documentation in the code itself? Does the code build straight out of source control, or do you first have to manually perform many small configuration tweaks before it will build? How is a release build made? Is it the same process as the development builds, or do you have to follow some very different set of steps? When the build runs, is it quiet, or are there many, many warnings?
- Look for tests. Are there any? How much of the codebase is under test? Do the tests run automatically, or do they require an additional build step? How often are the tests run? How much coverage do they provide? Do they appear appropriate and well constructed, or just a few simple stubs that appease the project mandate for tests? There is an almost universal link here: code with a good suite of tests is usually also well-factored, well-thought-out, and well-designed. These tests act as a great route into the code under test, helping you understand the code's interface and usage patterns. It's also a great place from which to start working on a bugfix (you can start by adding a simple, failing, unit test – then fix that test, without breaking the others).
- Look at the directory structure. Does it match the code shape? Does it clearly reveal the areas, subsystems or layers of the code? Is it neat? Are third-party libraries neatly separated from the project code, or is it all messily intermingled?
- Look for the project documentation. Is there any? Is it well written? Is it up to date? Perhaps the documentation is written in the code

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

itself using NDoc, Javadoc, Doxygen, or a similar system. How comprehensive and up-to-date does this documentation appear?

- Run tools over the code to determine the shape. There are some great source navigation tools available, and Doxygen can also produce very usable class diagrams and control flow diagrams.
- Are there any original project requirements documents or functional specifications? (In my experience, these often tend to bear little relation to the final product, but they are interesting historical documents nonetheless!) Is there a project wiki where common concepts are collected?
- Does the code use any specific major frameworks and libraries? You can't learn every aspect of all of them initially, especially since some libraries are huge (Boost, I'm looking at you). But it pays to get a feel for what facilities are provided for you, and where you can look for them.
- Browse through the code to get a feel for the low-level quality. Take a view on the level and quality of code comments. Is there much dead code – redundant code commented out but left to rot? Is the coding style consistent throughout? It's hard to draw a conclusive opinion from a brief investigation like this, but you can quickly get a reasonable feel for a codebase from such an investigation.
- By now you should be able to get a reasonable feel for the shape and the modularisation of the system. Can you identify the main layers? Are the layers cleanly separated, or are they all rather interwoven? Is there a database layer? How sensible does it look; can you see the schema? Is it sane? How does the app talk to the outside world? What is the GUI technology? The file I/O tech? The networking tech?

Learn by doing

A woman needs a man like a fish needs a bicycle. Programmers need bicycles far more. They make excellent metaphors!

You can read as many books as you like about the theory of riding a bicycle. You can study bicycles, take them apart, reassemble them, investigate the physics and engineering behind them. But you may as well be learning to ride a fish. Until you get on a bicycle, put your feet on the pedals and try to ride it for real, you'll never advance. You'll learn more by falling off a few times than from days of reading about how to balance.

It's the same with code. Reading will only get you so far. You can only really learn a codebase by getting on it, by trying to ride it, by making mistakes. By rolling your sleeves up and getting stuck in. Don't let inactivity prevent you from moving on. Don't erect an intellectual barrier to prevent you from working on the code.

I've seen plenty of great programmers initially paralysed through their own lack of confidence in their understanding.

Stuff that. Jump in. Boldly. Modify the code.

> The best way to learn code is to modify it. Learn from your mistakes.

So what should you do? Look for places where you can immediately make a benefit, but that will minimise the chances you'll break something (or write embarrassing code).

Aim for anything that will take you round the system.

- Try some smaller things, like tracking down a 'simple' bug, one that seems to have a very direct correlation to an event you can start hunting from (a GUI activity, for example). Start with a small, repeatable, low risk fault report, rather than a meaty intermittent nightmare.
- Run the codebase through some code validators (like Lint, Fortify, Cppcheck, FxCop, ReSharper or the like). Look to see if compiler warnings have been disabled; re-enable them, and fix the messages. This will teach you the code structure and give you a clue about the code quality. Fixing this kind of thing is often not too complex, but very worthwhile; a great introduction. It often gets you round most of the code quickly. Non-functional code changes around the system lead you to learn about how things fit together and what lives where. It will also give you a great feel for the diligence of the existing developers, and for which parts of the code are the most worrisome, and will require extra care when you handle them.
- Inspect the build system. Can you fix any build issues with it?
- Study a small piece of code. Critique it. Determine if there are weak spots. Refactor it. Mercilessly. Name variables correctly. Turn sprawling code sections into smaller well-named functions. A few such exercises will give you a good feel for how malleable the code is and how yielding to fixes and modifications. (I've seen codebases that really fought back against refactoring.)
- Look at the tests. Work out how to add a new unit test, a new test file to the suite. How do they get run?
- Do some spit-and-polish on the interface. Make some simple UI improvements that don't affect core functionality, but do make the app more pleasant to use.
- Gauge the quality of the source file organisation in the directory hierarchy. Does it match the layout in the project files? Move the files into a better place.
- Does the code have any kind of top-level README documentation files that explain how to start working on it? If not, create one and include the things that you have learned so far. Ask one of the more experienced programmers to review it – this will show how correct your knowledge is, and also help future newbies.
- As you are gaining understanding of the system, maintain a layer diagram of the main sections of code. Keep it up to date as you learn more. Do you discover that the system is well-layered, with clear interfaces between each layer and no unnecessary coupling? Or do you find the sections of code are needlessly interconnected? Look for ways of introducing interfaces to bring about separation without changing the existing functionality.
- Perform software archaeology on code that looks questionable. Drill back through source control logs and 'svn blame' (or the equivalent) to see who was responsible for some of the messes. Try to get a feel for the number of people who worked on the code in the past. How many of them are still on the team?

Conclusion

Scientific investigations are a sort of warfare carried on in the closet or on the couch against all one's contemporaries and predecessors

Thomas Young

The more you work on new codebases, the more you are able to pick up new code effectively, just as the more you exercise, the less pain you feel and the greater the benefit you receive. ■

Questions

1. Do you often enter new codebases? Do you find it easy to work your way around unfamiliar code? What are the common tools you use to investigate a project? What tools can you add to this arsenal?
2. Describe some strategies for adding new code to a system you don't understand fully yet. How can you put a firewall around the existing code to protect it (and you)?
3. How can you make code easier for a new recruit to understand? What should you do now to improve the state of your current project?
4. Does the likely time you will spend working on the code in the future affect the effort and manner in which you learn existing code? Are you more likely to make a 'quick and dirty' fix to code that you will no longer have to maintain, even though others will have to later on? Is this appropriate?

Hello World in JavaScript

Frances Buontempo demonstrates how to unit-test a simple JavaScript program.

Pete Sommerlad [1] mentioned a typical C++ **"Hello world"** program at the ACCU 2012 conference. Aside from being rude about the comments that automatically generated versions tend to end up with, I was struck by his point that usually the salutation is bolted straight into the **main** function, rendering it un-testable, other than by eyeballing the output. Many times, we start with a 'hello world' application when we learn a new language. Many times, we bolt the message straight into **main**, rendering it un-testable. This means people tend to resort to debugging by **printf**, or trying to use a debugger to see what's going on as they feel their way round a new language. In fact, printing **"Hello world"** in **main** is the *de facto* standard way to debug by **printf**. It has been around for years and is here to stay for a long time, if we don't shake things up a bit.

I'd like to buck this trend by presenting "Hello world" for JavaScript using test driven development. We shall start with a test that fails, then get it to pass, and might consider refactoring with the safety of the test. This way I have learnt how to write tests while I learn JavaScript, which I hope will make me more efficient, avoiding my frequent trick of trying to figure out what's going on with the moral equivalent of **printf** statements until I can't take it any more and do it properly.

So, where to start? With a unit testing framework, obviously. There seem to be a few for JavaScript, but I'll demonstrate jasmine [2].

We put our source somewhere, and our 'spec' somewhere. The spec describes the tests. We can pull this all together with one html file, which runs all the specs. Jasmine comes with a sample, including a `SpecRunner.html`, which we can copy and point at our "Hello world" code. I will follow the jasmine structure of having a source directory, `src`, and sibling `spec` directory.

Let's start with a failing test, given in Listing 1.

The function **describe** introduces a test suite, which takes free text as a name and a function implementing what must be tested. Inside the test suite, the spec is given via the **it** function, using **expect**. Again, this takes free text, making it easy to give clear descriptions of the expectation and good error messages if a test fails. By starting with a failing test, we can see how good the error message will be when a test fails. The `SpecRunner.html`, based on the one that comes with jasmine, wires the code and specs together, and points to the jasmine library. Notice that the comment is backwards; since it's open source, I could change it, but for now let's ignore the comments.

```
<!-- include source files here... -->
<script type="text/javascript"
        src="spec/HelloSpec.js"></script>

<!-- include spec files here... -->
```

Listing 1

```
describe("Hello world", function() {
  var greet;

  beforeEach(function() {
    greet = new Hello();
  });

  it("should say Hello world", function() {
    expect(greet.greet()).toEqual("Hello world");
  });
});
```

Listing 2

```
function Hello() {
}
Hello.prototype.greet = function() {
  return "Hello";
}
```

```
<script type="text/javascript"
```

Figure 1


```
src="src/Hello.js"></script>
```

The spec calls our code under test, `src/Hello.js`, shown in Listing 2. The code deliberately returns the wrong string, so we can start with a failing test. We get the output in Figure 1.

We now have a test, which fails. Step one complete.

We can now change the JavaScript code to say **"Hello world"** instead and we have a passing test (see Listing 3).

Listing 3

```
function Hello() {
}
Hello.prototype.greet = function() {
  return "Hello world";
}
```

We now have a test which passes. Step two complete.

The next step could be refactoring, for example I could simplify the code to stop using the prototype, but I'll leave it here for now to catch up on the mentored developers' JavaScript project. Before I do, notice we have a hello world function, and a test for it, but don't actually have a hello world app yet. However, I have a test spec that shows how to use my Hello function, so it's simple to write the main application.

We have seen how to write a "Hello world" application in JavaScript, using the unit testing framework Jasmine. Making sure we started with a failing test allowed us to verify a useful error message was produced on failure. Once the "Hello world" function was written, and the test passed, it was easy to write the actual "Hello world" application. This may seem over the top, but it is important to start as you mean to go on. This test setup provides a place for future JavaScript noodles, without resorting to debugging or inspecting print statements. There are several JavaScript testing

FRANCES BUONTEMPO

Frances has a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. She has been programming professionally for over 12 years. She can be contacted at frances.buontempo@gmail.com.

The Composition Pattern and the Monad

Richard Polton explains how Monads can reduce complexity.

In this instalment of our pattern application series we are going to revisit the COMPOSITION PATTERN and explore how it leads us to the MONAD PATTERN. Recall that COMPOSITION allows us to reduce repetition by constructing new functions from existing functions, instead of simply copying the common code into the body of the new function as might otherwise be the case. The MONAD PATTERN will allow us to extend what we have already learnt about COMPOSITION and, as we shall see below, to add, at the very least, decisions between the components of the composition as well as possible shared state information. The ultimate goal will be, as before, to remove or reduce repetition in our code by defining new reusable units.

For our worked example, suppose that we are presented with the following not particularly nice but not especially unusual piece of imperative code. For all the hyperbole, this particular example is not too bad having as it does only the one return value, albeit tucked away in the middle of the function, but it does help to demonstrate the essence of what we are about to explore. So, without further ado, consider Listing 1.

Ultimately we would like to separate the success branch and the failure branch of `ThisIsIt()` individually but the current construction makes this separation difficult. With this goal in mind, let us identify the individual components of the branches and extract them into functions.

Using our active imagination, and playing fast-and-loose with the types for the moment, let us extract each individual `if` statement into a separate function and rewrite `ThisIsIt` accordingly (see Listing 2). At this point a pattern can be seen emerging. Take a starting value, perform a check and then branch, returning the result of the branch as the result of the function. Okay, this could be defined recursively but we're going to tackle this iteratively. Also note that exceptions are used to break the control flow (we will come back to this at a later point). So, to consider the success branch, we take a value, check it, call a function, take the value, check it, call a function, take a value, check it ... and so on.

RICHARD POLTON

Richard has enjoyed functional programming ever since discovering SICP and feels heartened that programming languages are evolving back to LISP. He likes 'making it better' and enjoys riding his bike when he can't. He can be contacted at richard.polton@shaftesbury.me

Hello World in JavaScript (continued)

frameworks, but Jasmine was easy to get up and running and the ability to name your test suites in words rather than using underscores or camel case is beautiful. I hope this article reminds me, and others, to always write tests first. ■

References

[1] http://wiki.hsr.ch/PeterSommerlad/files/2012ACCU-1.pdf
[2] http://pivotal.github.com/jasmine/

Listing 3

```
<!DOCTYPE HTML PUBLIC
  "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
  <title>Hello world</title>

  <script type="text/javascript"
          src="src/Hello.js"></script>

  <script type="text/javascript">
    var greet = new Hello();
    document.write(greet.greet());
  </script>
</head>

<body>
</body>
</html>
```

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

Listing 1

```
RetType ThisIsIt()
{
  var a = GetInitialValueFromSomewhere();
  if( a!=null)
  {
    var b = DoSomethingWith(a);
    if(b!=null)
    {
      var c = DoSomethingWith(b);
      if(c!=null)
      {
        var d = DoSomethingWith(c);
        if(d==null)
           throw new ArgumentException("Arg!!!");
        return d;
      }
      else
      {
        throw new ArgumentException("Bad C");
      }
    }
    else
    {
      throw new ArgumentException("Bad B");
    }
  }
  else
  {
    throw new ArgumentException
       ("It's all gone pear-shaped");
  }
}
```

So far so good. It's abundantly clear where the return value is coming from although you'll have to take my word for it that we can consider the failure cases using this approach. Anyway, we now have a load of temporary variables which serve to do little more than take up screen space (and the compiler will assuredly 'file' them away) so let's replace them with

```
var value = DoIt4(DoIt3(DoIt2(DoIt1(
  GetInitialValueFromSomewhere())))));
```

Hmm, it's all back-to-front. Let's suppose we can re-arrange the functions in their natural order with a helpful **Extension** method which we'll call **Then** (and yes I've changed **GetInitialValueFromSomewhere** into **GetInitialValue** because they're not quite the same thing, as we'll see shortly) then we can write a function **value_fn** such that

```
Func<???,???> value_fn = GetInitialValue()
  .Then(DoIt1)
  .Then(DoIt2)
  .Then(DoIt3)
  .Then(DoIt4);
```

This example of the FLUENT PROGRAMMING style is also essentially the COMPOSITION pattern, so called because we can compose a single function from two or more functions. If we name our resultant composition **JustDoIt**, then for some types **A** and **B** to be defined, we can write it as

Currying

We can always transform a function taking *N* parameters into a sequence of functions taking one parameter each by writing (this is known as 'currying', see previous article in CVu and also see http://en.wikipedia.org/wiki/Currying)

```
Func<T1,Func<T2,RetType>> transformedFunction =
  (T1 t1) => (T2 t2) => originalFunction(t1,t2);
```

and as an aside this might be worthy of consideration when ordering functions parameters – keeping those that change least often on the inside, ie **t1** in this example, and so in the context of **JustDoIt** we might call the function in this manner

```
t2.In(transformedFunction(t1).Then(...))
```

Listing 2

```
RetType ThisIsIt2()
{
  var a = GetInitialValueFromSomewhere();
  var b = DoIt1(a);
  var c = DoIt2(b);
  var d = DoIt3(c);
  var e = DoIt4(d);
  return e;
}

B DoIt1(A a)
{
  if(a!=null)
  {
    return DoSomethingWith(a);
  }
  else
  {
    throw new ArgumentException
       ("It's all gone pear-shaped");
  }
}

C DoIt2(B b)
{
  if(b!=null)
  {
    return DoSomethingWith(b);
  }
  else
  {
    throw new ArgumentException("Bad B");
  }
}

D DoIt3(C c)
{
  if(c!=null)
  {
    return DoSomethingWith(c);
  }
  else
  {
    throw new ArgumentException("Bad C");
  }
}

RetType DoIt4(D d)
{
  if(d!=null)
    {
      return d;
    }
    else
    {
      throw new ArgumentException("Arg!!!");
  }
}
```

```
Func<A,B> JustDoIt = x =>
  (DoIt1.Then(DoIt2).Then(DoIt3).Then(DoIt4))(x);
```

enabling us to rewrite the entire expression as

```
Func<???,???> value_fn
  = GetInitialValue().Then(JustDoIt);
var value = value_fn();
```

That's much better, and with appropriate and meaningful naming, the essence of the algorithm can be captured succinctly. The key here is that **Then** does not actually execute the **DoIt[1-4]** functions itself. It is

instead the classic composition operator (little 'o') as described in a previous CVu article

```
static Func<T1,T3> Then<T1,T2,T3>
  (this Func<T1,T2> f, Func<T2,T3> g)
{
  return t1=>g(f(t1));
}
```

As you have probably observed, the manner in which the parameter is passed to `DoIt1` in the definition of `JustDoIt` does not fit this pattern neatly because the parameter has to be passed explicitly, but we can always choose to maintain the symmetry of the functions by writing

```
Func<A,B> JustDoIt = x => x.In(DoIt1
  .Then(DoIt2)
  .Then(DoIt3)
  .Then(DoIt4));
```

where `In()` is the classic method–parameter inversion function which was also discussed in a previous CVu article, ie

```
static T2 In<T1,T2>(this T1 t1, Func<T1,T2> f)
{
  return f(t1);
}
```

and using `In` again, we can write

```
var value =
  GetInitialValueFromSomewhere().In(JustDoIt);
```

At this point we do have a workable solution, as we have captured the individual pieces of the failure branch in our `DoIt[1-4]` functions. However, there is a further factorisation we can make before we can say 'Done'. We just need to deal with the failure cases in a way which does not obscure the success branch of our algorithm. If all the `DoIt[1-4]` functions had only a single branch, ie no `if` statement at all, then the type of `JustDoIt` would be simply `Func<T1,RetType>`. However, there is at least a binary decision which we need to capture. As it happens, it turns out that what we are trying to achieve is well-known in certain circles ;-) and there is, therefore, an established pattern, the MONAD PATTERN, which we can utilise. In the MONAD PATTERN a number of operations are chained together in some order with the possibility of some state being transported between the operations behind the scenes. In our specific case, the state that is transported is the union of the result of the binary decision (null or not null) and the accompanying data value. In the example here, we are going to pull the success or failure check out of the `DoIt` functions into a wrapper which we shall call `Bind()` so that the `DoIt[1-4]` functions can then concern themselves exclusively with the success branch.

Now, using a new function called `ToMonadicType` which turns a plain value into a 'decorated', ie monadic, value we rewrite our functions as in Listing 3, where `Bind` is defined as Listing 4.

We can think of our `Bind` function as a generalised `Then` function which encapsulates the binary decision thus allowing us to focus on the success branch in our `DoIt` functions. In Monad terms, `ToMonadicType` is referred to as `Return`. This particular Monad is known as the EXCEPTION MONAD. If the evaluation is successful then it stores a value and if unsuccessful then it stores the exception information.

We can, of course, also implement a monadic `JustDoIt` function, say `JustDoItM`, and use it in `ThisIsIt3` should we desire to do so. In this case, we would write

```
Func<A, RetType> JustDoItM = x=>x.ToMonadicType()
                             .Bind(DoIt1M)
                             .Bind(DoIt2M)
                             .Bind(DoIt3M)
                             .Bind(DoIt4M);
```

and then, in `ThisIsIt3`, we would write

```
var a = GetInitialValueFromSomewhere()
        .In(JustDoItM);
```

Listing 3

```
MonadType<B> DoIt1M(A a)
{
  return DoSomethingWith(a).ToMonadicType();
}
MonadType<C> DoIt2M(B b)
{
  return DoSomethingWith(b).ToMonadicType();
}
MonadType<D> DoIt3M(C c)
{
  return DoSomethingWith(c).ToMonadicType();
}
MonadType<RetType> DoIt4M(D d)
{
  return d.ToMonadicType();
}
RetType ThisIsIt3()
{
  var a =

GetInitialValueFromSomewhere().ToMonadicType()
    .Bind(DoIt1M)
    .Bind(DoIt2M)
    .Bind(DoIt3M)
    .Bind(DoIt4M);
  if(a.HasValue) return a.Value;
  throw a.ShowException;
}
```

So, to summarise, the steps we have taken in this example are:

- Obtain some evil imperative code with lots of nested if .. else clauses
- Separate into functions each of which should return the value that is to be used as the input parameter to the next function in the sequence
- Define `Return`
- Define `Bind` and move the decision point, eg the 'if null' checks, into the `Bind`
- Redefine the separate functions in terms of the Monadic Return and process only the success branch
- Compose the functions.

Do we really need all those exceptions?

If we take a mental step back and reconsider the code, then we can see that the exceptions in `ThisIsIt` are being used purely to break the flow, almost in a `goto` sense. While this is quite normal, the raising of an exception in this manner is not strictly an exceptional circumstance. It is, however, a common idiom. The use of this idiom, though, forces consumers of our code to speak exception or die (to paraphrase the 'Stormtroopers of Death', see http://en.wikipedia.org/wiki/Speak_English_or_Die). That is to say that we are enforcing the decision that the branching can only be between a successful transformation and a catastrophic failure. Back in the day, we would have written a `return` statement instead of the `throw ...`, returning some default value which the consuming code knew needed to be handled in a special way. Of course,

Listing 3

```
static MonadType<U> Bind<T,U>
  (this MonadType<T> t, Func<T,MonadType<U>> f)
{
  if(t.HasValue)
  {
    var value = f(t.Value);
    return value!=null ? value : new MonadType<U>
       (new ArgumentException("Bad input data"));
  }
  return new MonadType<U>(t.ShowException);
}
```

Listing 4

```
var input = new[]{a1,a2,a3,a4};
var output = new List<T>();
foreach( var elem in input)
{
  try
  {
    var tmp = ThisIsIt(elem);
    output.Add(tmp);
  }
  catch(ArgumentException arg)
  {
    ; // continue
  }
}
```

it is part of modern style to replace these early returns with a thrown exception without pausing for thought.

Therefore, it can be seen that replacing the original code with the monadic code presented above has the effect of allowing the consumers to make their own decisions about what constitutes a fatal error in the context of their own code, and all without having to reserve a magic value.

There is in common usage another very similar Monad called the OPTION MONAD which could also be used here. Given that the exceptions are being used as **goto** statements, we could use the OPTION MONAD instead because the OPTION MONAD does not store anything in the failure case. **System.Nullable** and OPTIONTYPE from a previous article are approximations to the OPTION MONAD.

However, let us expand the example slightly to illustrate a common usage of the OPTION MONAD. Suppose that we have a sequence of input data and we wish to transform it into another sequence containing the same number elements or fewer using **ThisIsIt** as the transformation function. As **ThisIsIt** stands, we would have to write something like Listing 4.

Yuck! Nestled deep in that code is the actual transformation but you would have to look carefully to be sure that nothing else was happening. This is, of course, similar to the pattern that **map**, also known in C#-land as **Enumerable.Select**, was designed to implement. The difficulty we will experience here if we tried to use **map** is that **map** guarantees the number of output elements will be the same as the number of input elements.

Of course, we could compose a filter and a map to transform our input sequence but as this is a common pattern let us define a function, called **Choose**, as shown in Listing 5.

We can now write the sequence transformation as

```
var output = input.Choose(ThisIsit);
```

This has neatly removed the exception handling code from the front-line, as it were, but has the unfortunate side-effect of polluting what could otherwise be a generic function, **Choose**, with a specific mechanism for

Listing 5

```
static IEnumerable<U> Choose<T,U>
  (this IEnumerable<T> input,
   Func<T,U> transformation)
{
  foreach(var elem in input)
  {
    try
    {
      var intermediate = transformation(elem);
      yield return intermediate;
    }
    catch(Exception)
    {
      ; // continue
    }
  }
}
```

Listing 6

```
static IEnumerable<U> Choose<T,U>
  (this IEnumerable<T> input,
   Func<T,Option<U>> transformation)
{
  foreach(var elem in input)
  {
    var intermediate = transformation(elem);
    if(intermediate.IsSome)
      yield return intermediate.Some;
  }
}
```

Listing 7

```
var output = input.Choose(elem =>
{
  try { return ThisIsIt(elem).ToOption(); }
  catch(Exception) { return Option<U>.None; }
});
```

handling the failure branch. Instead, with the appropriate definition of **Option**, we can raise the failure handler out of the generic function and write **Choose** as Listing 6 and the sequence transformation as Listing 7.

In this example **Option<U>** is the OPTION MONAD. Let's use the **Choose** transformation with the current implementation of **ThisIsIt**.

This has considerably reduced the noise in our code and has made it much easier for the reader to determine the intent. Of course, if we can rewrite **ThisIsIt** as **ThisIsIt4** returning an OPTION MONAD instead of an EXCEPTION MONAD or throwing an **Exception**, we can rewrite our transformation as

```
var input = new[]{a1,a2,a3,a4};
var output = input.Choose(ThisIsit4);
```

which has completely reduced the code to its fundamental behaviour.

So, what is a Monad? For the technical definition, you need look no further than the Wikipedia article http://en.wikipedia.org/wiki/Monad_(functional_programming). You could also take a look at the F# wikibook (http://en.wikibooks.org/wiki/F_Sharp_Programming/Computation_Expressions), which gives a good description too, as does the book Learn You A Haskell For Great Good (http://learnyouahaskell.com/a-fistful-of-monads).

Additionally, having got to grips with the idea, you could do worse than take a look at http://lorgonblog.wordpress.com/2007/12/02/c-3-0-lambda-and-the-first-post-of-a-series-about-monadic-parser-combinators/ for a relatively easy-to-follow practical demonstration of another common monadic pattern.

Additionally, Luca's blog post http://lucabolognese.wordpress.com/2012/11/23/exceptions-vs-return-values-to-represent-errors-in-f-iithe-critical-monad/ is worth a read. However, if you are coming at this from an OOP perspective, these articles might still not help all that much.

Recall that Monads originate in functional programming languages. As the description implies, functional programming languages are about functions. Monads give functional programmes the ability to bundle data with functions, as a better closure if you will, or maybe like a structure or class in OOP terminology although the MONAD PATTERN is still useful in OOP because it provides a standard pattern that can be used to describe our classes. Strictly not as a technical definition of a Monad but instead a woolly and loose description, I find it helps to think of Monads as a container of functions which satisfy a well-known interface, a structure if you will. In addition, a Monad may also contain some state information, ie data. In OOP terminology we can think of a Monad as a Class or more accurately a family of Classes that implement a specific interface. This interface, which we shall think of as the Monadic interface, as a minimum supplies **Return** and **Bind**.

These two functions have well-defined signatures, which could be written in C# syntax as

```
M<T> Return<T>(T t)
```

and

```
M<V> Bind<U,V>(M<U> u, Func<U,M<V>> fn)
```

where **M** is the Monad in question. Return, then, is an adapter which transforms an unadorned instance of **T** into a Monad of **T**. **Bind** is a wrapper around a transformation between underlying types.

Return is probably more familiar if it were written

```
static M<T> ToMonadOfT<T>(this T t)
```

but **Bind** is more interesting. **Bind**, which we saw earlier in this article, is where the real work is done.

That is, the body of **Bind** is the code which makes the decisions about how the monad should behave. So, let us consider a Monad as a structure with a well-known interface which groups a number of functions together possibly with some state information. Often the functions are grouped together in a 'workflow', which may have an implied or actual order. We can think of workflow as a form of functional composition.

As monads are so useful, it has come about that compilers for certain programming languages 'know'about the Monadic interface and so they can provide special syntax for structures which satisfy the base requirements, eg the F# compiler transforms **Return** and **Bind** to **return** and **let!** respectively.

Two commonly occurring monads, and among the simplest, are the OPTION MONAD and the EXCEPTION MONAD which we have explored already.

Both of these essentially encapsulate the same decision, albeit with less or more additional information respectively. That is, both of these Monads encapsulate a binary choice. The Monad is either 'true' (in the context of the monad) and contains some state information or 'false'. The difference between these two Monads is that, in the 'false' case, the EXCEPTION MONAD contains additionally some exception information.

Trivial partial implementations are shown in Listing 8 and Listing 9.

Let us now consider some familiar use-cases of the EXCEPTION MONAD.

Listing 8

```
class Option<T>
{
  public Option(T t) {_t = t; _isSome=true; }

  public T Some {
    get {
      if(_isSome) return _t ;
      else throw new AccessException();
    }
  }
  public Option<T> None {
    get { return Option(); }
  }
  public bool IsSome { get { return _isSome; } }
  public bool IsNone { get { return !IsSome; } }
}
```

Listing 9

```
class MException<T>
{
  public MException(T t)
    {_t = t; _isSome=true; _exception=null; }

  public T value {
    get {
      if(_isSome) return _t ;
      else throw new AccessException();
    }
  }
  public bool HasException {
    get { return _isSome; }
  }
}
```

1. Dictionary look-up

A fairly common device occurring in various projects is when we want to look up some data based upon some key, which might not be present. One of the simplest examples of this is looking up some value in a dictionary. In C#,

```
var value = dict["key"]
```

fails with an exception if **key** is not present in the dictionary. (As an aside, this is the reason for the existence of the **TryGetValue** function.)

In general, therefore, we cannot use the natural syntax to query the dictionary unless we choose to wrap the query in a **try .. catch** block. And so suddenly our simple and clear one-liner turns into this

```
string value;
try
{
  value = dict["key"];
}
catch(Exception ex)
{
  value = string.Empty;
}
```

Yuck! The essence of what we are trying to do has been completely obscured by the packaging. In a manner reminiscent of the **Functional.Switch** from a previous article, we could define **Try** such that

```
var value = Functional.Try<Exception>
   (()=>dict["key"], ex=>string.Empty);
```

where the first function parameter is a lambda function that should be 'tried' and the second is a lambda function that should be evaluated in the event that an exception of the specified type is caught. This is a bit clunky though, as it does not map cleanly to common usage, ie **try { ... } catch(exception) { ... }**. Instead the exception type is bundled with the **Try()** signature and the **Catch** clause must be presented either as a lambda, which isn't too bad, or an array of lambdas in the event that we wish to handle one of multiple exception types, which is less good.

Even this, though, has demerits because it requires a magic value of **string.Empty**. It might be the case that it is meaningful to store **string.Empty** in the dictionary but now we have effectively reserved it as an error indicator. Step forward the EXCEPTION MONAD, which we will call **MException** for the purposes of this article. First we define **Return** trivially as

```
static MException<T> ToMException<T>(this T t)
{
  return new MException<T>(t);
}
```

and **Bind** as

```
static MException<V> Bind<U,V>
  (this MException<U> u,
   Func<U,MException<V>> fn)
{
  if(u.HasException)
    return new MException<V>(u.ShowException);
  try
  {
    return fn(u.Value);
  }
  catch(Exception ex)
  {
    return new MException<U>(ex);
  }
}
```

With these two functions we can write the slightly long-winded

```
var value = "key".ToMException()
    .Bind(key => dict[key].ToMException());
```

where value is of type **MException<U>** and so we have one line again. At this point, you are probably wondering what exactly has been gained especially as we have had to create two new functions and repeatedly call one of them. The primary difference between the monadic code above and the simpler functional code earlier is that the monadic code stores extra information. That is the state of the monad is either the value or the exception that was thrown. In order to access the value of **value**, we would call **value.HasException** and, if no exception is present, **value.Value**.

Recall that **Bind** operates like the short-circuit operators. That is, if the condition is false then it returns early and so any subsequent **Bind** operations will also terminate early. Suppose then that we have a chain of transformations. In the fluent style we might write

```
var value = "key".ToMException()
                 .Bind(Lookup.Then(Validate)
                 .Then(Enrich).Then(toEx));
```

where

```
Func<K,V> Lookup = key => dict[key];
Func<V,MException<V>> toEx = v => v.ToMException();
```

and **Validate** and **Enrich** are defined appropriately.

There is a problem with this, though. The **dict** is bundled inside **Lookup** as a closure variable. So we rewrite our fluent one-liner slightly using

```
LookupIn(dict).Then(Validate).Then(Enrich)
              .Then(toEx)
```

where

```
Func<Dictionary<K,V>,Func<K,V>> LookupIn
  = dict => key => dict[key];
```

Alternatively we could encapsulate the inner binder function as a single function

```
Func<Dictionary<K,V>,
Func<K,MException<Enrichment>>> Generate =
dict => dict.In(LookupIn.Then(Validate)
              .Then(Enrich).Then(toEx));
```

And so we have

```
var value =
  "key".ToMException().Bind(dict.In(Generate));
```

We might also want to extend this to other aspects of the **Dictionary**. For example, consider the extension method **ToDictionary**. This takes a sequence as its input and returns a new **Dictionary**. However, **ToDictionary** behaves badly if there are duplicates. In fact, the entire operation fails in this instance which, if the key or value creation functions are expensive, could be undesirable. Instead we could use the monad pattern, again the EXCEPTION MONAD, to make our lives easier. I leave this as an exercise for the reader (although sample code can be found on Google Code in functional-utils-csharp).

2. Try...Catch...Finally

As another example taking the EXCEPTION MONAD as a starting point we can implement functional wrappers for the decidedly-imperative **try...catch...finally** construct. Listing 10 presents a trivial implementation of what is essentially three further versions of the **Bind** function for **MException**, calling them **Try**, **Catch** and **Finally**.

As can be seen, the signatures are not precisely monadic and this is because of the usage pattern but the monadic behaviour can be seen clearly. As **Try** returns a monad, it follows that it can be passed to a following **Catch** or **Finally**. Note that the implementation of **Finally** given here is not strictly a **Bind** implementation because it is really a wrapper around an **Action** but, in order to be able to chain these functions together into a single statement, it is necessary that **Finally** should be a proper function and, in order to be a function, it must return something, so we choose to return the monad itself. From a semantic point of view, chaining multiple **Finally** function calls is of questionable value but, in the above trivial implementation, it is possible. That aside, the functional composition of the monadic pattern means that we can chain together multiple **Catch** functions in a single clause, leading to code of the form

Listing 10

```
public static class TryCatchFinallyMonadBuilder
{
  private static MException<T>
     Return<T>(this T t)
  {
    return new MException<T>(t);
  }

  public static MException<U>
     Try</*T,*/U>(Func</*T is void,*/U> tfm)
  {
    try
    {
      return tfm().Return();
    }
    catch (Exception ex)
    {
      return new MException<U>(ex);
    }
  }

  public static MException<T> Catch<Ex, T>
    (this MException<T> tcf,
    Func<Ex, T> catchClause)
    where Ex : Exception
    {
      return tcf.HasException &&
      ((tcf.ShowException is Ex) ||
        tcf.ShowException.GetType()
       .IsSubclassOf(typeof(Ex)))
       ? catchClause
       (tcf.ShowException as Ex).Return()
       : Return(tcf.Value);
    }

    public static MException<T>
       Finally<T>(this MException<T> t,
                  Action<T> tfm)
       {
         tfm(t.HasException ? default(T)
             : t.Value);
         return t;
       }
}
```

```
var r = Try(()=>DoSomethingAndReturnANewU()).
  Catch((AccessException acc) =>
     new U("Caught an AccessException")).
  Catch((Exception ex) =>
     new U("Caught an Exception")).
  Finally(CleanUp);
```

To conclude, the OPTION and EXCEPTION MONADS both enable us to encapsulate decisions and as a consequence can be used to implement short-circuit logic in a way which allows clean separation between the success branch from the failure branch. The MONAD PATTERN itself is a very useful meta-pattern which can be used in many situations where we have a sequence of functions we wish to apply in a certain order. Additionally, if there are data to be transported between the functions which do not contribute to the algorithm then these can be bundled within the Monad. The two Monads which have been introduced here are among the simplest and there are, therefore, many more of greater complexity. Feel free to trawl the internet looking for other monads, and see if you can work them into your code. ■

Code Critique Competition 79

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

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

Last issue's code

I thought it was time to start trying out some of the new C++11 concurrency features; but it looks like the compilers might be a bit buggy still. Shouldn't the following program cleanly separate log output from the two threads? I started with MSVC 2012 and it worked Ok until I turned on optimising; so I tried g++ with -std=c++11 – both sometimes give mixed lines.

Sample output is in Listing 1 and the code is in Listing 2.

Listing 1

```
29 Sep 23:40:37 [1]: starting the calculation
29 Sep 23:40:37 [2]: starting the calculation
29 Sep 23:40:37 [1]: next...
29 Sep 23:40:37 [2]: next...
29 Sep 23:40:37 [129 Sep 23:40:37 [2]: next...
]: next...
29 Sep 23:40:37 [2]: next...
29 Sep 23:40:37 [1]: next...
29 Sep 23:40:37 [2]: next...
29 Sep 23:40:37 [1]: next...
29 Sep 23:40:37 [2]: ending the calculation
29 Sep 23:40:37 [1]: next...
29 Sep 23:40:37 [1]: ending the calculation
```

Listing 2

```
#include <iostream>
#include <mutex>
#include <string>
#include <thread>

// get now
std::string now()
{
  time_t const timeNow = time(0);
  char buffer[16];
  strftime(buffer, sizeof(buffer),
    "%d %b %H:%M:%S", localtime(&timeNow));
  return buffer;
}
```

Listing 2 (cont'd)

```
// log message
void log(int id, std::string const & message)
{
  std::mutex mutex;
  mutex.lock();
  std::cout << now() << " [" << id << "]: "
    << message << std::endl;
  mutex.unlock();
}

// simulate some activity
void dosomething()
{
  int i(0);
  for (int j = 0; j != 10000; ++j)
  {
    for (int k = 0; k != 10000; ++k)
    {
      ++i;
    }
  }
}

// calculate
void calculate(int id, int count)
{
  log(id, "starting the calculation");
  for (int idx = 0; idx != count; ++idx)
  {
    dosomething();
    log(id, "next...");
  }
  log(id, "ending the calculation");
}

int main()
{
  try
  {
    std::thread t1(calculate, 1, 5);
    std::thread t2(calculate, 2, 4);
    t1.join();
    t2.join();
  }

  catch (std::exception const & ex)
  {
    std::cerr << "exception: " << ex.what()
      << std::endl;
  }
}
```

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

Critiques

Peter Sommerlad <peter.sommerlad@hsr.ch>

I am not sure I am getting all problems, but I can point out an explanation for the behaviour.

1. The synchronization is wrong

To synchronize two threads with a mutex they must share the same object to synchronize with. That means, in many systems such mutex variables are more or less global. In the situation of the example code with functions only, the mutex might be passed down the call chain from main or be a global variable (arrgh).

The following code will use a new mutex instance whenever it is called thus in fact not doing any cross-thread synchronization at all:

```
void log(int id, std::string const & message)
{
  std::mutex mutex; // new object each time
  mutex.lock();
  std::cout << now() << " [" << id << "]: "
    << message << std::endl;
  mutex.unlock();
}
```

In addition it lacks RAII, so if the output would result in an exception (i.e. `std::bad_alloc` from `std::string`'s construction) the lock wouldn't be released, if the mutex object really would be shared across the threads.

Minor issues:

- code assumes at least 32-bit int, because it increments to quite a large number in the `dosomething` function, an overflow would cause undefined behaviour. I would also prefer using an `unsigned` type to avoid any undefined behaviour even in case of overflow.
- time output C++11: The time output is still very c-ish. A C++11 version of the log would use `std::chrono` elements (unfortunately the following code is untested, since my compiler doesn't implement `std::put_time` manipulator yet, which works like `strftime()` but more efficient by directly using the underlying `streambuf` as buffer and also without the danger of a too short buffer.

```
using clock=std::chrono::system_clock;
auto tm=clock::to_time_t(clock::now());
// create a C struct time_t
std::cout <<std::put_time(std::localtime(&tm),
  "%d %b %H:%M:%S"));
```

- `main` could use a try-catch function block.

2. Why it works on some systems anyway

Because `std::cout` is a global object, using it concurrently would be dangerous anyway. While `iostream` objects in general are not safely to be used from multiple threads, `std::cout` normally is. `std::cout` is usually synchronised with C's stdio library `stdout` stream, and because of the following clause [iostream.objects.overview] of the standard is synchronized, and then it is safe to use it from different threads (it won't crash).

> 4 Concurrent access to a synchronized (27.5.3.4) standard iostream object's formatted and unformatted input (27.7.2.1) and output (27.7.3.1) functions or a standard C stream by multiple threads shall not result in a data race (1.10). [Note: Users must still synchronize concurrent use of these objects and streams by multiple threads if they wish to avoid interleaved characters. - end note]

The note also tells us what to do about interleaving characters that the original code's poster asked about. But why does it work on some platforms? That is actually a quality of implementation issue or sheer luck :-). An implementation might create a temporary object on the first call of the shift operator that will lock the stream, holding the lock while the expression statement is active and release the temporary object after the full expression is done, thus effectively releasing the lock.

3. How it could be done right

Pass the mutex's reference through the call chain. Because of thread's constructor we might need to use `std::ref()` as a wrapper to pass the mutex reference instead of a copy, which is not possible. In the `log()` function I used `std::lock_guard<std::mutex>` to actually do the locking the RAII/Scoped Locking Idiom way as it always should be done. An even better version would be to make all functions taking the log a template function where the mutex's type is the template parameter. I also C++11-ified the `now()` function a bit, but only so that I could still compile it on my system. It might not be the best version, but at least IMHO a working one.

```
#include <iostream>
#include <mutex>
#include <string>
#include <thread>
#include <chrono>
std::string now() {
  using clock=std::chrono::system_clock;
  auto const timeNow =
    clock::to_time_t(clock::now());
  // no put_time yet: should be better
  char buffer[16];
  strftime(buffer, sizeof(buffer),
    "%d %b %H:%M:%S", localtime(&timeNow));
  return buffer;
}
void log(int id, std::string const & message,
         std::mutex &mutex) {
  std::lock_guard<std::mutex> locker(mutex);
  std::cout << now() << " [" << id << "]: "
    << message <<std::endl;
}
// simulate some activity
void dosomething() {
  unsigned long i(0);
  for (int j = 0; j != 10000; ++j) {
    for (int k = 0; k != 10000; ++k) {
      ++i;
    }
  }
}
void calculate(int id, int count,
               std::mutex &lock) {
  log(id, "starting the calculation", lock);
  for (int idx = 0; idx != count; ++idx) {
    dosomething();
    log(id, "next...", lock);
  }
  log(id, "ending the calculation", lock);
}
int main()
  try {
    std::mutex lock;
    std::thread t1(calculate, 1, 5,
      std::ref(lock));
    std::thread t2(calculate, 2, 4,
      std::ref(lock));
    t1.join();
    t2.join();
  }
  catch (std::exception const & ex) {
    std::cerr << "exception: " << ex.what()
      << std::endl;
  }
}
```

Joe Wood <joew60@yahoo.com>

There are a number of problems with the supplied code, which we shall tackle in such a way as to get a better program after each iteration, although your view on the order may vary.

First the reason that the program causes interleaved output. The mutex in `log` is declared `std::mutex mutex`, this causes a new mutex to be created for each invocation of `log`, with the result that there is no locking

between threads since each thread locks on its own mutex. The cleanest solution is to change the declaration of mutex to `static std::mutex mutex`, thus providing a single mutex encapsulated in the `log` function. The resulting program now works as desired, but we can do better.

If an exception is thrown in `log`, the mutex is left locked and the system will deadlock. In the sample program this is not a major problem because it will have failed anyway, but in a production system we might not want to leave a locked mutex. This is easily overcome by adding the line

```
std::lock_guard<std::mutex> lock(mutex);
```

immediately below the declaration of mutex, and removing the line

```
mutex.unlock();
```

at the end of `log`. `lock_guard` guarantees (even in the presence of exceptions) to release the mutex when the log function terminates.

There are potentially a few problems with the function now. C++11 introduces `nullptr` explicitly to distinguish between "0", "0l" and `NULL` in various forms. Therefore it would be better to replace the line

```
time_t const timeNow = time(0);
```

with

```
time_t const timeNow = time(nullptr);
```

`buffer` is declared as being 16 `char`s long, which is fine provided the "%b" conversion parameter to `strftime` never returns more than three characters. I am not sure if this is true in all locales, so let us err on the side of caution and make buffer `25 char`s long.

The function `localtime` uses a global buffer which is not thread safe. As currently written this is not a problem as now is called from inside `log` and hence guarded by the mutex. However, we might want to call `now` from outside `log`, so we better use `localtime_r` and supply our own `struct tm` buffer. Hence the call to `strftime` now becomes

```
struct tm tm_time;
strftime(buffer, sizeof(buffer),
  "%d %b %H:%M:%S",
  localtime_r(&timeNow, &tm_time));
```

Which brings use neatly to a potential performance issue. We definitely want to write each output line without any interleaving, but holding the mutex while we build up the output reduces the scope for exploiting multi-processors. Provided we can accept that the displayed time may not be exactly when the output line was produced, we can get better multi-processor usage by replacing the `log` function with these two functions.

```
// lock output stream and display message
void put_message (std::string const &message){
  static std::mutex mutex;
  std::lock_guard<std::mutex> lock(mutex);
  std::cout << message << std::endl;
}

// log message
void log(int id, std::string const &message) {
  std::stringstream oss;
  oss << now() << " [" << id << "]: "
    << message;
  put_message(oss.str());
}
```

Regretfully, I do not know of any way to test the resulting program beyond trial and error. I am not sure if I like the free inter mixing of C++ threading primitives into any old code, at least in Ada tasking primitives have to be confined to tasks. This example looks straight forward with a protected type (light weight Ada task which guarantees mutual exclusion) to hold the `put_message` routine.

Huw Lewis <huw.lewis2409@gmail.com>

The mixed up output is down to a simple misunderstanding of mutex objects. The developer doesn't appear to realise that a mutex object must be shared by all threads accessing the shared resource (in this case the shared resource is `std::cout`). The `log` function declares a mutex object to perform this role, but as it is declared on the stack in this position it is an entirely new and independent (and therefore ineffective) mutex object for each call to `log`.

A single word fix is available – declare that `std::mutex` object as static. Now all calls to `log` from whatever threads will use the same mutex object. Job done? Yes, but no. That solution is crude and is vulnerable to race conditions in the initial construction of the mutex object.

I prefer the OO approach – let's make the `log` an object with a built in mutex to provide thread safety. Note also that I've used the `std::lock_guard` to lock and unlock the mutex using RAII.

```
// A log class that writes its
// output to std::cout.
class Log
{
private:
  // A mutex to protect std::cout from
  // concurrent use
  std::mutex mutex_;

  // get time now
  std::string now() const
  {
    time_t const timeNow = time(0);
    char buffer[16];
    strftime(buffer, sizeof(buffer),
      "%d %b %H:%M:%S", localtime(&timeNow));
    return buffer;
  }

public:
  // log a message to cout
  void log(int id, std::string const& message)
  {
    std::lock_guard<std::mutex> guard(mutex_);

    std::cout << now() << " [" << id << "]: "
      << message << std::endl;
  }
};
```

The worker thread functions could use a global `Log` object, but I think it would be neater to similarly wrap the `calculate` method up as a `Calculator` class with a `calculate` method. The `Calculator` constructor can take a reference to the `Log` object and use it throughout its lifetime.

There are many other options around the threading model. For example, the `log` could run in another thread with a mutex and semaphore protecting a queue of messages to be written by the thread's main loop.

Finally before I get carried away with over the top designs, I'd like to revert to a simpler scheme that better meets the requirement. The `std::cout` streaming operators are (for most compilers) 'atomic' operations i.e. each streaming operation on that object (`cout`) will not be interrupted by others in other threads. So it is possible to fix this without needing a mutex object!! The simple solution is to construct the string to be logged (e.g. using `ostringstream`), then send it to `std::cout` in a single operation.

```
// log a message to cout
void log(int id, std::string const& message)
{
  // format the string to be logged
  std::ostringstream output;
  output << now() << " [" << id << "]: "
    << message << std::endl;

  // send to cout in one go - no need to
  // synchronise!
  std::cout << output.str();
}
```

Ola M <aleksandra.mierzejewska@gmail.com>

[Solution from Ola + Michal]

In the given solution the threads are not synchronised, because the mutex is created locally in the `log` function. This means each thread will lock and unlock its own mutex.

To fix the problem, we could move out the mutex to the global scope or make it `static`. We chose however to pass the mutex as an argument. It's important to note here, that arguments to thread function are copied, so to pass the mutex by reference it must be wrapped in `std::ref`-attaching code.

Another thing that might need to be added is exception handling inside the calculate function – this is because exception thrown by that function would not be passed back to main thread.

```
#include <iostream>
#include <mutex>
#include <string>
#include <thread>

// get now
std::string now()
{
  time_t const timeNow = time(0);
  char buffer[16];
  strftime(buffer, sizeof(buffer),
    "%d %b %H:%M:%S", localtime(&timeNow));
  return buffer;
}

// log message
void log(int id, std::string const & message,
  std::mutex & mutex)
{

// static std::mutex mutex;
  mutex.lock();
  std::cout << now() << " [" << id << "]: "
            << message << std::endl;
  mutex.unlock();
}

// simulate some activity
void dosomething()
{
  int i(0);

  for (int j = 0; j != 10000; ++j)
  {
    for (int k = 0; k != 10000; ++k)
    {
      ++i;
    }
  }
}

// calculate
void calculate(int id, int count,
  std::mutex & mut)
{
  //std::mutex & mut = *m;
  log(id, "starting the calculation", mut);
  for (int idx = 0; idx != count; ++idx)
  {
    dosomething();
    log(id, "next...", mut);
  }
  log(id, "ending the calculation", mut);
}
```

```
int main()
{
  try
  {
    std::mutex mut;
    std::thread t1(calculate, 1, 5,
      std::ref(mut));
    std::thread t2(calculate, 2, 4,
      std::ref(mut));
    t1.join();
    t2.join();
  }
  catch (std::exception const & ex)
  {
    std::cerr << "exception: " << ex.what()
      << std::endl;
  }
}
```

Commentary

I read the code that inspired this critique in a recently published book; this goes to show you shouldn't apply example code, no matter what the source, without thought!

As all the entrants pointed out the problem is that the mutex object must be shared between all the threads that access the shared resource (`std::cout`) and this means it cannot be an automatic variable as separate instances of this are created on each call to the function.

Getting synchronisation right is hard. The simple code shown here worked fine on a multi-core machine when compiled using MSVC without optimisation; merely executing concurrent code successfully – even several times – does not prove correctness.

Note too that although making the mutex `static` is guaranteed by the C++11 standard to be safe not all compilers fully support this behaviour (MSVC, for example, is not in general race-free for local static variable initialisation.)

In this case there is *another* problem with synchronisation – we can fix the bug in the `log` function but we cannot fix the problem that *other* pieces of the program may also use `std::cout` and we may have interleaved characters from this concurrent use. Joe's solution of using `ostringstream` may improve matters, if used consistently, but this will depend on the actual implementation of the stream.

Lawrence Crowl has submitted a paper (N3395) to the ISO C++ committee about this very issue; to quote:

> At present, stream output operations guarantee that they will not produce race conditions, but do not guarantee that the effect will be sensible. Some form of external synchronization is required. Unfortunately, without a standard mechanism for synchronizing, independently developed software will be unable to synchronize.

The paper proposes a few possible solutions; as yet no clear decision for the best way forward has been reached.

The other issue in the code is the use of a fixed-size buffer for the call to `strftime` when using one of the string output formats: in this case `%b`. There is no guarantee that I can find about the maximum (or minimum) length of the abbreviated month name in the different locales a C runtime may support.

So, for example, if I add this call at the start of `main`:

```
setlocale(LC_TIME, "frs");
```

then I get no timestamps in my log output as the abbreviation for Décembre is "déc." which, being four letters long, means the call to `strftime` fails as the output would exceed `sizeof(buffer)`. (Here I'm using the MSVC runtime on Windows.)

The Winner of CC 78

All four entrants successfully identified the problem caused by multiple instances of the mutex variable and provided various solutions. However

Joe was the only person to pick up the questionable call to **strftime** so I think he is the worthy winner of this issue's critique.

Code critique 79

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

Thanks to Francis Glassborow for sending me this one: modified to prevent an easy Google search for other people's comments.

I keep getting a compiler warning when I compile the code with gcc. Please help me get rid of it!

```
gcc cc79.c -o cc79
cc79.c: In function 'addFirst':
cc79.c:36:17: warning: assignment from
incompatible pointer type [enabled by default]
cc79.c: In function 'printList':
cc79.c:46:9: warning: assignment from
incompatible pointer type [enabled by default].
```

I've tried with Microsoft VC and that gives me an even more confusing error:

```
cc79.c(36) : warning C4133: '=' : incompatible
types - from 'LLNODE *' to 'LLNODE *'"
```

The code is in Listing 3.

(As usual, there's the 'presenting problem' to solve but there are other issues to highlight in the code.)

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

Listing 3

```
/* This program builds a basic linked list. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct { // Define a linked list node
  char *name;
  struct LLNODE *next;
}LLNODE;

void addFirst(char *data); //Declare function
                           //prototypes
void printList(void);

LLNODE *head = NULL; //Define global pointer
                     //variable head

int main(int argc, char *argv[])
{
  addFirst("Peter");
  addFirst("Paul");
  addFirst("Mary");
  printList();
  return EXIT_SUCCESS;
}
void addFirst(char *data)
{
  LLNODE *newNode;
  newNode = malloc(sizeof(LLNODE));
  newNode->name = data;
  newNode->next = head; //errors here
  head = newNode;
}
void printList(void)
{
  LLNODE *itr = head;
  while(itr != NULL)
  {
    printf("%s\n", itr->name);
    itr = itr->next;  //And errors here.
  }
}
```

Regional Meetings

Chris Oldwood rounds up a whole series of talks from ACCU London, and Paul Grenyer gives us SyncNorwich.

ACCU London

Chris Oldwood provides a summary of the happenings at three recent ACCU London meetings.

October 2012: Continuous Delivery – compèred by Ed Sykes

Continuous Delivery is one of those techniques that I know what it is in theory but aren't sure what everyone actually does in practice. One obvious way to find out more about it would be to buy *the* book on the topic – *Continuous Delivery*. Instead I chose a different route – to listen to various people describe how they're doing it.

The venue was the Mozilla offices near Leicester Square. This was a new venue to me and it was a real treat to be greeted by hot pizza, and fizzy pop courtesy of 7Digital/1E. The turnout was excellent with around 50 people spread across the chairs and sofas. Most of those were different faces, but there were a still a few regulars to catch up with before the session.

Rather than a single speaker or set of lightning talks the format was three speakers each giving closer to a 15-minute presentation on how they are executing Continuous Delivery in their organisation. With a large audience clearly wanting to extract every last ounce of information from the presenters there was a barrage of questions after each segment that they deftly handled.

they walk-the-walk as well as talking-the-talk

One of my goals was to find out what the size of a 'feature' is so that when I see the stream of tweets from companies boasting at the number of features they've pushed into production today I have some frame of reference. Luckily I didn't have to wait too long as first up was Chris O'Dell describing how the process works at 7Digital. Her team seems to have adopted many of the modern practices, such as pair programming and Kanban to create a very fluid environment. This wasn't plain sailing and they clearly still have some baggage to address for some time yet. The most useful part of the presentation was the slides that walked through a 'day in the life' of a change, from development right through to deployment. Chris left you with a feeling that they walk-the-walk as well as talking-the-talk.

Next up was Ed himself with Cosmin Onea to explain the 1E approach. This is a company that seems to have to deal with a little more 'paperwork' for the final hurdle which I can definitely empathise with. They had a video similar to Chris's earlier slides that presented the change workflow, but this time in a style more reminiscent of a production-line. Naturally there were many parallels to the first talk, with automated build, testing and deployment the common thread. A desire to avoid feature branches and focus on small well defined tasks was also apparent.

The final speaker was Robert Chatley, who instead chose to talk about how he has tackled cloud based deployments. I suspect a fair portion of the audience is working on systems that are, or intend to, run in the cloud and so this was probably highly relevant for them. The crux of the presentation was about how to keep that purity of deploying exactly what you've tested, which in his case extended right down into the OS image on which the tests were run. He also provided some tips on how you might go about switching over from one version of the software to the next which requires a fair bit of thought when you have a considerable number of nodes to upgrade and whilst maintaining a minimum level of capacity.

Nice though the offices of Mozilla are with its alluring fridges, it was time to switch to a more traditional watering hole and continue the discussion with proper ale. Not unsurprisingly I discovered there is quite a bit of latitude in the definition of Continuous Delivery and that for some this only extends from DEV through to UAT. Sadly there is often some heavy-handed 'review' process in place to bring the pipeline to an eventual standstill. Even so, I came away positive knowing that my team and I are heading in the right direction and that there is still a wealth of tooling to explore to help extract every last ounce out of the process.

November 2012: Introduction to Clang – Dietmar Kuhl

The London branch of the ACCU went back to its roots last night at the Bloomberg offices as Dietmar Kuhl gave an introductory talk on Clang. This is one of those technologies, along with LLVM, that has been floating around my subconscious just waiting to be investigated further; particularly as I've been playing with GCC purely for its static code analysis benefits.

Dietmar started with a very brief overview of what Clang is, might be used for, and what some of the goals were that shaped its architecture. The most notable of these are the avoidance of exceptions and RTTI, although it has its own variant of RTTI instead. He also mentioned its open source nature and the fact that it has active input from a couple of big names which is always reassuring.

We discovered that Bloomberg use other mainstream C++ compilers for its core business and that Dietmar's interest in Clang is more on the research side at present. After the brief introduction he swiftly moved on to showing us how to build a plug-in for Clang. These are how you tap into the various pipelines within the tool so that you can execute some of your own code to run on the various events as they occur during compilation.

Obviously such a large tool can't be covered in any depth in just 60 minutes, but he did a great job of picking a realistic static code analysis problem and then showed how you might go about implementing it. The example was about detecting dodgy casts (C style and the C++ reinterpret kind) from long to int, which can break when moving from 32-bit to 64-bit platforms. The amount of code was surprisingly small and most of the time was probably spent explaining how types were represented and then how you go about comparing them correctly.

One of the goals of Clang is better diagnostics, which for those of us who remember using deciphering tools like STLFilt on their C++ errors, will surely appreciate. He showed the output from his example plug-in and it was good to see the types referred to by their local typedefs rather than the underlying names.

CHRIS OLDWOOD

Chris started as a bedroom coder in the 80s, writing assember on 8-bit micros. Now it's C++ and C# on WIndows. He is the commentator for the Godmanchester Gala Day Duck Race and can be reached at gort@cix.co.uk or @chrisoldwood

PAUL GRENYER

Paul Grenyer is a husband, father, software consultant, author, testing and agile evangelist. He can be contacted at paul.grenyer@gmail.com

With a 50+ audience there were plenty of questions afterwards as Dietmar had clearly stirred imaginations. The most notable question for me was about how this might affect the vendors of lint-style code analysis tools as the platform looks to provide plenty of scope in this area – if the developer community is willing. The question Dietmar clearly wanted to be asked was 'which pub?' and with that we trotted off to The Flying Horse opposite to further our discussions.

Clearly the local brew was good stuff because it seems at one point I suggested you might be able to use Clang as a code generator that would allow you to write C++ 11 style code that could then still be compiled with Visual C++ 6. But it was just a joke, right?

December 2012: Christmas Dinner

There are only two times of the year when the ACCU London branch doesn't meet for a traditional presentation – April, when the conference is on, and Christmas. Instead we get together at a restaurant and spend the entire evening chewing-the-fat (quite literally in some cases) rather than it being confined to a swift pint after an hour's lecture. The format is still largely the same though as we retire to a pub afterwards. Oh, and we still end up in the same chain of pizza restaurant each year.

You'd think that such a progressive bunch of programmers with one eye on the agile landscape would embrace change and mix it up once in a while. But then, why bother. The restaurant is just a means to an end, one in which we get to catch up with the physical forms of some of those people we only normally recognise by their tiny avatars (or an egg) and converse with in chunks of 140 characters or less. Despite modern gadgets allowing us to stay 'in touch' with one another 24x7 there are still times when only beer and pizza will do.

SyncNorwich 6 Review

Paul Grenyer reviews a recent SyncNorwich gathering sponsored by Aviva.

Aviva Christmas Special at Carrow Road

SyncNorwich [1] is going from strength to strength. A year and a day before the Aviva Christmas Special at Carrow Road, I was sat in the Coach and Horses [2] on Thorpe Road for the very first Agile East Anglia [3] meeting. A year ago there were half a dozen of us, tonight, after the addition of Norwich Startups and Norwich Developer Community and rebranding as SyncNorwich in June, there were 110 of us in one of the most prestigious venues in Norwich. It's difficult to describe how incredible that feels, so I won't try, anyone who was there will have seen how much it meant to me.

Tonight's event was sponsored by Aviva. SyncNorwich is very grateful to them for hiring the venue, buying a drink for everyone and three fantastic speakers.

First up is Juliana Meyer who gives an introduction to SyncNorwich for those who are new to the group and a recap of many of the events that SyncNorwich has been involved with over the last six months. I have seen variations of this presentation many times, but Juliana always makes it feel fresh and new.

Juliana was followed by the charismatic John Marshall from UK Trade and Investment [4], who uses his two minute presentation judiciously to tell SyncNorwich about the companies he is working with and the money he has to give away to help companies trade overseas. Tonight he is just as informative and entertaining as at Hot Source [5] a week ago.

Next up we had SyncNorwich favorite, ignite style lightning talks [6]. This time around there were four. Chris Leighton was up first and despite her slides working against her gave an excellent and well delivered presentation on startup training from the Business Skills Clinic [7]. Chris was followed by local entrepreneur Keith Beacham who took us through a very slick presentation about the highs and lows of startup companies and investment. I first saw William Harvey give his lightning talk on funding for low carbon companies at SyncNorwich Corner at the Common Room [8] in November. It was great to have him at a full SyncNorwich event and soak up some of the infectious enthusiasm. Last up were Tom

McLoughlin and Josh Davies from FXHome [9]. They gave us a hilarious description of the trials and tribulations of making their startup successful and told us about some of the amazing people in the video industry that they've worked with.

Before the break SyncNorwich gave FXHome the chance to show their latest promotional film to the group. It was fantastic and you can watch it at http://www.retinaburnblog.co.uk/whats-your-idea-fxhome-hitfilm/.

After the break it was the turn of Rob Houghton whose charismatic and engaging Northern style was the highlight of the evening and went down extremely well with the SyncNorwich crowd. He told us about Horses Vs Tanks and his vision to revolutionise the use of technology at Aviva. Rob's session was followed by an extremely interesting, amusing and informative question and answer session.

The last of the presentations came from Jason Vettraino and Jason Steele who spoke about the joys and perils of a mobile application that they developed at Aviva. This was the first technical presentation we've had at SyncNorwich since Dan Wagner-Hall from Google spoke about testing [10]. It was great to have some good honest technology back in the programme and again this presentation was both amusing and informative and followed by an engaging question and answer session with Sky Viker-Rumsey providing his usual difficult questioning to the Jasons.

The Aviva SyncNorwich Christmas special was wrapped up with a run down of future events, including SyncConf [11] given by John Fagan and our January Student/Employer Speed Dating & CyberDojo [12] event given by Seb Butcher. After that were the Smart421 [13] robots! The highlight that many had come for.

References

[1] http://www.syncnorwich.com/
[2] http://www.thecoachthorperoad.co.uk/
[3] http://paulgrenyer.blogspot.co.uk/2012/12/agile-east-anglia-short-history.html
[4] https://www.ukti.gov.uk/
[5] http://hotsourcenorwich.co.uk/
[6] http://en.wikipedia.org/wiki/Ignite_(event)
[7] http://businessskillsclinic.com/
[8] http://paulgrenyer.blogspot.co.uk/2012/11/syncnorwich-at-common-room-review.html
[9] http://fxhome.com/
[10] http://paulgrenyer.blogspot.co.uk/2012/10/syncnorwich-4-reviews.html
[11] http://syncconf.com/
[12] http://www.syncnorwich.com/events/87908322/
[13] http://www.smart421.com/

Acknowledgements

Thanks to James Neale (www.jamesnealephotography.com) for kind permission to use the photograph.

Standards Report

Mark Radford reports the latest from C++ Standardisation.

When the first C++ standard was published in 1998, the standards' committee then spent a lot of time and effort on the issues raised as a result. These issues consisted of reports (from the user community) of potential defects in the standard, or simply requests for clarification regarding various aspects of the standard. When I wrote my last column (focusing on the Portland meeting) I was very enthusiastic about the work going into new C++17 features, and I rather glossed over the issue processing aspect of the committee's work. Ever since the C++2011 standard was published last year, once again, much effort has gone into (and continues to go into) processing the many issues.

Up to now I've been writing on the basis that C++17 is the next version of the standard. However, it seems there has been a slight variation in this plan. An issue processing exercise also took place following the publication of the 1998 standard, the result being 'C++2003', a 'bug fix' update to the standard. The same thing is being planned now: an update, known as C++14, is now scheduled for 2014. The difference this time is that C++14 is likely to contain a small number of new features.

What new features will go into C++14 is, at the moment, still unknown. However, a decision is expected to be taken at the Chicago meeting (October 2013). Basically, anything that is ready by then is likely to be included. Likely candidates include: user-defined literals (N3402), the **`get<TYPE>`** part of tuple tidbits (N3404) and **`string_ref`** (N3442). Mike Spertus' paper (N3404) entitled 'Tuple Tidbits' caught my eye, so I'll pick on this paper to talk about a little more.

Currently fields in tuples are accessed by their index e.g. **`get<2>()`**. Mike Spertus' idea is for them to be accessible by their type, so that **`get<std::string>`** would access a field of type **`std::string`**. Of course this would only work if the tuple contained only one field of type **`std::string`**. The idea is that if there were more than one then a compile error would result.

Another very neat idea in this paper (sadly, as I currently understand it, not to be in C++14) is 'functoriality'. The idea is that a tuple object with fields that are pointers to functions, or function objects, can be applied to another tuple object containing data fields, in one operation. I'll repeat an example from the paper (using simple function pointers) to give you the idea:

```
auto funcs = make_tuple(sqrt, strlen, atof);
auto vals = make_tuple(9, "foo", "7.3");
auto result = funcs(vals); // result is
  // tuple<double, size_t, double>(3, 3, 7.3)
```

Seeing this it occurred to me that these three lines of code could be compressed into a single statement:

```
auto result = make_tuple(sqrt, strlen,
  atof)(make_tuple(9, "foo", "7.3"));
```

Now, if we have a function pointer called **`execute`** and another called **`on_args`**, that each point to the appropriate **`make_tuple<>`** instantiation, the following would be very expressive indeed:

```
auto result = execute(sqrt, strlen, atof)
  (on_args(9, "foo", "7.3"));
```

You can find all of the standards committee's papers (past and present) at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/.

MARK RADFORD

Mark Radford has been developing software for twenty-five years, and has been a member of the BSI C++ Panel for fourteen of them. His interests are mainly in C++, C# and Python. He can be contacted at mark@twonine.co.uk

Two Pence Worth

An opportunity to share your pearls of wisdom with us.

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 have captured some of those gems and printed the best ones.

"Write TODOs in the code to save money on bug tracking software" Arthur, Arundel

"Make Friday 'Open Source' day and let your devs work on OSS projects. They don't do any work anyway" B., Bristol

"When pasting code from a book add some error handling to ensure it works correctly" Dr Love, London

"Use the Singleton pattern only when it would actually be of some benefit" Chris, Cambs.

"Liberally spread platform-specifics throughout your code – especially word size reliance – to show how much you know about your computer" Ed, London

"Write multiple statements on one line to make use of the extra screen width that multiple monitors provide" Arthur, Arundel

"Save money on heating bills by writing all your code to target GPGPUs" Chris, Cambs

"Brighten up your colleagues' day by using Unicode variable names" Anon

"Help save the planet by removing all unnecessary white space in code, thereby reducing file sizes" Ed, London

"Add some light-heartedness to your team by adopting Comic Sans as the in-house font" Chris, Cambs

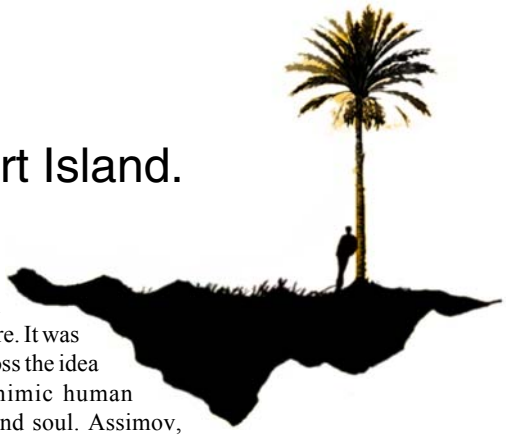
"Reduce finger strain by only using identifier names from keys in the middle row" Mavis, Beaconsfield

"Remove the X,C and V keys from developers' keyboards to stop them copy-and-pasting code" Chris, Cambs

If you have your own 2p to add to the collective wisdom of the group, send it to cvu@accu.org.

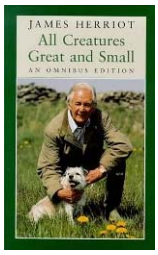
Desert Island Books

Ed Sykes finds things to read on a Desert Island.



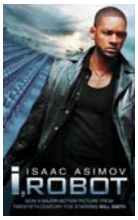
Ed's been a regular at the ACCU Conference and the mailing lists for quite some time, although I can't quite put my finger on exactly how long. I think my first recollection of him at the conference was ear-wigging on a conversation about unit testing and build management – topics which I think remain close to Ed's heart! Recently he's been getting involved in organising the ACCU London meetings, which has given me more opportunity to chat with him over a beer or two afterward, which is the usual form for these things.

All Creatures Great and Small



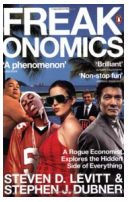
I love this semi-autobiographical work written - under the pseudonym James Herriot – by James Alfred Wight. I've read this book more than any other, mostly due to obsessive reading and re-reading as a pre-teen. One story in particular sticks: a collie who rampantly terrorises Herriot – a vet visiting the farm – by leaping through a hedge and gleefully barking in his ear. Set in the 40s and 50s it describes a time of change in the agricultural communities of Yorkshire, a change away from using animals to mechanisation. It vividly describes characters such as the Farnham brothers, each – in their own way – as mad as a cat in a bath. The book delights with comic observations and big heart. Marooned, this is the book that I would turn to first. Lying far away from England – alone amongst the palm trees – this book would provide home-comfort through widescreen panoramas of England's landscape and national character.

I, Robot



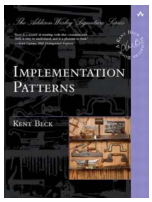
Isaac Assimov is a personal hero. His writing style is simple but his ideas are extraordinary. Looking back and connecting the dots, it planted the idea of studying Psychology and Artificial Intelligence at University – guiding me to programming. Without this book I would have studied German and French; I probably wouldn't be in London or writing software. *I, Robot* influenced other science fiction writers and thinkers. It wove The Three Laws of Robotics into the fabric of human consciousness and culture. It was the first time I came across the idea that software could mimic human intelligence, emotion and soul. Assimov, and his approach to life, has been a big inspiration to me and this was the first of his works I read.

Freakonomics



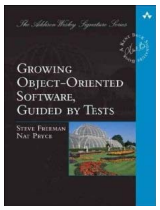
Studying psychology showed the power of statistics in answering interesting questions. *Freakonomics* blew my mind. Here, data revealed stories of the human condition hitherto untold. Amongst them, the startling story between abortion and crime rates. It opened my eyes to a new side of economics, gently lighting a path of interest towards classic economics. Lucky enough to live in London, I saw the authors talking about their follow-up, *SuperFreakonomics*, at the LSE. I even had a chance to ask them a few questions and get my books signed. Magical.

Implementation Patterns



There are lots of books about patterns at the OO level, but this book is about small patterns that emerge at the micro level. One idea that stuck: pay attention to the symmetry in your code, it tells you about the design. Reading it felt like being in Kent's head. This book brought maturity and awareness of my own decision making at the statement, expression, function and class level. The strength of this book is that it taught me to think about thinking, a skill that goes beyond java – the language of the examples.

Growing Object Orientated Software Guided By Tests



This book is about the London school of TDD. It's the story of the creation of a software system using TDD at the unit and system level. It named themes and expressed idea that were appearing in my own work: TDD moving to the system level; The trend towards automation; the value in faking interfaces to external systems. Particularly rewarding was the mentored group on the accu-mentored mailing list. The authors joined the study group, clarifying and expanding their thinking. I think this is destined to become a modern classic.

What's it all about?

Desert Island Books is based (loosely) on the popular BBC Radio 4 programme, Desert Island Disks (http://www.bbc.co.uk/radio4/factual/desertislanddiscs.shtml). Many ACCU members have chosen their Desert Island Books, and there are plenty more to go. If you would like to share your Desert Island Books, please email cvu@accu.org

Choose 4 'technical' books – books that have influenced your programming life or that you would like to read – and explain what attracts you to them. Include a novel and two albums – you can slip in a film if you want – as this helps us get to know you better as a person.

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

Bookcase

The latest roundup of book reviews.

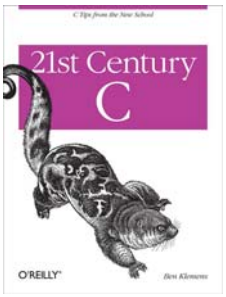
If you want to review a book, your first port of call should be the members section of the ACCU website, which contains a list of all of the books currently available. If there is something that you want to review, but can't find on there, just ask. It is possible that we can get hold of it.

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

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

Jez Higgins (jez@jezuk.co.uk)

21st Century C



By Ben Klement, published by O'Reilly, ISBN 978-1-4493-2714-9

Reviewed by Alexander Demin

This is the first sentence from the preface: 'C has only a handful of keywords and is a bit rough around the edges, and it rocks. You can do anything with it. Like the C, G and D chords of a guitar, you can learn the basic mechanics pretty quickly, and then spend the rest of your life getting better.'

Frankly, I had been hooked by this book from the start because I personally share exactly the same feeling about C. So, I continued reading, and here is my, of course, biased and subjective review.

As follows from the title, this book is about the latest cutting edge C, but it is not a textbook. Instead, the author explains some details of modern C, which many people still do not use sticking instead with 'classic' approaches, and also covers best practices for productive working with C.

First, the author briefly covers the situation with standards (C89, C99 and C11) and major implementations (gcc, clang). He steps through programming environments of modern C, debugging and documenting, version control, and packaging. Then he comes to the language itself. There is a chapter called 'C syntax you can ignore', which I read first. The rest of the book is dedicated the best practices: use of structures and pointers, text (mostly about Unicode), object-oriented approaches in C and a few mainstream libraries.

It may look like the material is a bit chaotic, but, again, this is not a textbook. The material definitely helps with rethinking your current use of C applying all new fancy bells and whistles.

Finally, a couple of my personal findings. For example, I had not heard at all about the `restrict` keyword and the `asprintf()` function. I liked a lot that the author promotes makefiles instead of piling up zillions of scripts, plus the advice of using `private_` prefix in structure field names (simple and harmless but practical).

Any book can become tedious if you agree with everything. But I found that some topics like 'a good use of `goto`', autotools and doxygen generate a lot of arguments in me. Such 'holy war' subjects make the book even better because you will have more topics for a chat with your colleagues on the kitchen.

The Economics of Software Quality

By Capers Jones and Olivier Bonsignour, published by Addison-Wesley, ISBN 978-0-13-258220-9

Reviewed by Paul Floyd

There are two authors that get referenced a lot in the body of literature on software development when it comes to measuring software. They are Barry Boehm and Capers Jones. That's not a large number, I guess because not many organizations have the means to do that sort of research and even fewer are willing to make it public. Whilst Boehm worked in the US defence industry, Capers Jones worked at IBM.

Starting on the positive side, this book covers all aspects of quality and in depth. For metrics, it does preach for the use of function points rather than lines of code. Probably the most interesting parts for me were the measurements of risks of defects by development stage (this somewhat assumes a waterfall-y development process) and the comparison of effectiveness of various methodologies and tools.

There is quite a lot of repetition in the book. It reminded me a bit of a politician being interviewed, when the politician is repeatedly making a point even if the interviewer is trying to move on to something else. This does get the point over, particularly if you're picking the book up and trying to use it to look up a reference. When I read it cover to cover though, it did drag on a bit. I wasn't too keen on the tables of subjective measurements given numerical values to 2 decimal places, like 'Risk of application causing environmental damages – 9.80'. I feel that this gives a false sense of precision and authority. There were a few clangers that undermine this authority. For instance, Perl is written Pearl, Mac is written Mack and one that really set my teeth on edge (wearing my Electronics Engineer hat) was the claim that 'Electrical engineering uses K to mean a value of 1,028, but software engineering uses K for an even 1,000'. Yes, that is 1,028, not 1,024. They are both even, though you might say that 1,000 is more of a 'round' figure. And no, Electrical engineers never use 1,024 for K for electrical things. 1Kohm or 1KV is always 1,000. When it comes to addressing memory then the 2^{10} version is often used. Ah, bee in bonnet. Another electronics oddity crops up in a table of languages that do not support static analysis, which includes Verilog (a hardware description language that has a great deal of static analysis tool support).

One of the points repeatedly made is that the cost of fixing a defect does not change depending on when it is discovered in the development cycle, contrary to the popularly held belief that the later a defect is found, the more it costs to fix. I can accept this if the defect is a coding defect. I don't agree if the defect is in the design error, which in my experience takes much longer to fix than something like an out-by-one coding error. Another point that is strongly emphasized is the increase in risk (and the number of defects) with the size of a project. I have no beef with that.

In summary, an interesting book that is ironically a bit let down by a few defects.

View From the Chair

Alan Griffiths
chair@accu.org

When it comes to delegation there's a big difference between responsibility and authority. In theory one cannot delegate responsibility – even when someone in authority 'makes someone else responsible' for something they retain the responsibility for what happens. We see this in the news week after week with politicians, heads of businesses and other organisations being held responsible for what someone else actually did (or didn't) do. If something goes wrong 'on their watch' they are ultimately responsible.

Authority is different: it can, and should, be delegated. If the management decide a computer system needs to be developed then they delegate the authority to a development team that is more able to focus on the details. But the management remains responsible for the actions of the group they've appointed.

Of course, it is equally true that responsibility can be accepted. That development team accepts responsibility for making the decision – but that is a new responsibility (to the management) not a shift in the responsibility the management has to shareholders and customers.

Why am I going on about this?

Because it applies to the ACCU.

The membership has delegated the authority to run the organisation to me and the other members of the committee.

As a committee we too have delegated authority to various committee and non-committee members, special interest groups and contracted work to non-members.

But despite delegating authority to people and groups that accept responsibility we all retain responsibilities that cannot be delegated. You,and other members have a responsibility for what the committee has been doing on your behalf!

In my reports 'From The Chair' I've detailed a lot of what's been going on – both so that members get to see how the committee exercises its authority and meets its responsibilities, and also so that the people doing the work get recognition for the things they are doing. (In this regard I should mention that Martin Moene made and implemented some suggestions for improving the website – thanks Martin.)

For the same reasons the secretary has been posting committee minutes to the accu-members mailing list. You might not be interested in everything – but if there is something of interest you should be able to identify what progress is being made.

You can also see which committee members take responsibility for 'actions' and the manner in which those 'actions' are fulfilled.

It is for you, the members to decide whether the committee as a whole, and as individuals, has been doing a good job. And now is the time to start thinking about it – because if you want something to change you must act before the AGM.

There is one further thing that requires your urgent attention: the secretary has posted a draft for a revised constitution on the website and invited discussion on accu-members. At the time of writing the silence has been remarkable.

I don't intend to denigrate the work done by those drafting the new constitution – they've done a lot of work on a hard problem, and I think they've done a creditable job and special thanks are due to Mick Brooks and Giovanni Asproni. But changing the constitution like this is a BIG DEAL, it involves a number of compromises and I don't believe that the first draft can have encountered universal approval. It is not my experience that ACCU members are reluctant to express opinions. Where are the questions and suggestions?

Now is the time to raise and fix any issues you see with the proposals.

The deadline is rapidly approaching where a motion to change the constitution must be notified to the membership – after which the discussion is limited to voting 'yes' or 'no'. It would be very disappointing if the work done on constitutional change went to waste because no-one raised a legitimate concern early enough for it to be addressed,

Please everyone, have a look at the draft for a new constitution, have a look at what the committee has been doing and let us know whether you approve.