The magazine of the ACCU

www.accu.org

{cvu}

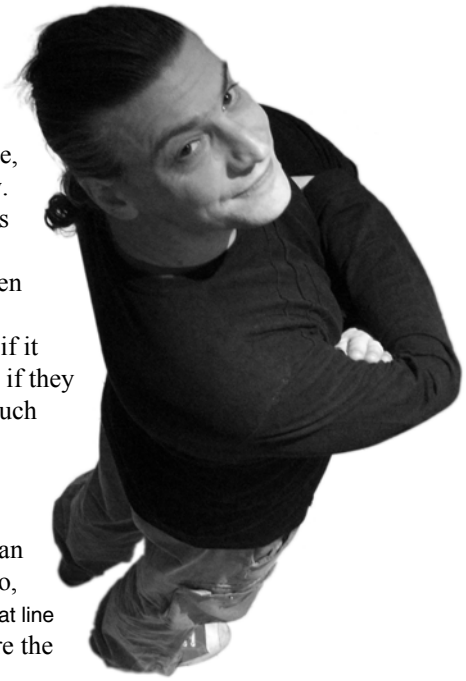
Volume 24 • Issue 1 • March 2012 • £3

Features

Regulars

{cvu}
Volume 24 Issue 1
March 2012
ISSN 1354-3164
www.accu.org

Too Clever By Half



It is almost certainly the result of watching too much, often bad, Science Fiction, but I was recently wondering about the phenomenon of computers having personalities.

I'll confess, I've been rediscovering *Blake's 7* of late, in which a computer called Orac appears frequently. It is a powerful super computer, and often proclaims itself far too busy with important matters to worry about the well- being of its human colleagues. It even occasionally refuses to answer direct questions if it considers them beneath its capabilities – especially if it thinks the humans could figure it out for themselves if they tried harder. Will computers of the future develop such characteristics?

I sometimes wonder if they already haven't. I quite often speak to my computers (Ok, shout is perhaps more accurate) but don't really get a response. Not an actual voice, in any case. The computers of today do, however, 'answer back' in some ways: "Syntax error at line 40: ; expected" isn't exactly a wise-crack, but I'm sure the sentiment is similar....

Compilers are prepared to do so much on our behalf. In C++, C#, and I'm sure others, the compiler will write an entire class for you, if you just type a lambda expression. So whycan't it just put the missing semi-colon in? It's clever enough to tell you – sarcastically – that it's missing.

The future is here, I tell you. Keep a close eye on the next generation of compilers for your favourite language, in case it's got cleverer than you. Of course, if it has, it might be clever enough to prevent you from being able to tell.

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 24.2: 1st April 2012
C Vu 24.3: 1st June 2012

Overload 109: 1st May 2012
Overload 110: 1st July 2012

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

A Book Turned Me Into A Programmer

Alexander Demin shares his relationship with the book that got him started.

In the career of any programmer, quite often there is something, like a spark or lightning, which gets you on a hook, kicks off your internal motor, and you have been drawn into software engineering, exploring this new universe nonstop. You just cannot stop.

For me it was a book. I know, it is very difficult to believe that one book can influence an entire career, but I'll try to convince you that it can.

Back in 1978, a book was published, called *Etudes for programmers* by Charles Wetherell. I couldn't read it that time because I was only one year old, so I eventually read it in 1990. It was an edition in Russian published in 1982.

As you probably figured out from the title, the book contains a list of problems, called etudes, to teach programming in real situations.

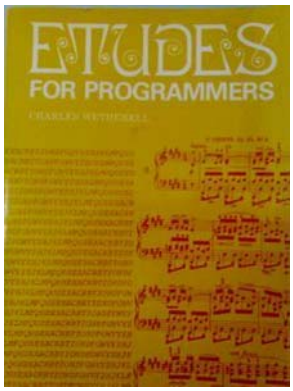
I'll stop bothering you with my memories and simply list topics covered by the etudes:

- Machine simulation (a game of 'Life', a business management game and a highway traffic simulation)
- Graphs and computer graphics (map colouring, puzzle construction and creation of mazes)
- Automatic text formatting and a format scanner
- Quine, a program printing its own sources
- Financial calculations (a home accounting system and calculations of investment yields)
- Turing machine simulator
- Data compression
- Optimal game strategies (Kalah and Mastermind)
- Searching for patterns among primes
- Solitaire statistics analysis
- Symbolic algebra package
- Numbers (errors using floating point and high-precision routines)
- Cryptanalysis (cracking a Vigenere cipher)
- Simulation of a large computer
- A linking loader
- A compiler for an algebraic language
- An interpreter or an interactive symbolic language

If we step back and review this list again, does it look very good as a CS course? Indeed.

When I read it for the first time, I was able to understand absolutely nothing. But the book described different aspects of programming so attractively, that it had inspired a vast interest in me to implement all the etudes, some time. I was stuck with the book and with programming for years.

Each etude has an introduction where the author places you in a situation. For example, you as a team leader have found a piece of paper on your desk with weird text. You realise that this is encrypted text. Is someone from your team stealing from you and trying to reveal all the killer features of your brand new secret compiler, and this is a message? To prove it and save the project you must crack the cipher. Off you go, welcome to the Vigenere cipher cracking problem.



After such intros you just jump on a problem ready to play the etude.

The etudes are designed to be implemented from scratch (except maybe the compiler of an algebraic language). The author gives you an idea to attack an etude, recommendations, orchestration (in fact, a language) and play time. When the author recommends a programming language for a particular etude, options look quite awkward today, for example, SNOBOL, XPL, BLISS, COBOL or FORTRAN. But surprisingly, all these modern fast and furious animals like C++, Java, Python, Ruby etc. don't really help to solve the problems faster. The problems still stand!

only by working on real problems can you develop as a programmer

At the end of the book, there are two author solutions: map colouring and data compression. They were implemented in FORTRAN and XPL. These sources are proof of a statement, that a bad language is not an excuse to write bad code. These two were clean and easy to understand.

There was an interesting story about the Russian edition of this book. The etude about cracking a cipher had an encrypted text. Obviously, it was in English. Russian editors wanted to publish a purely localized version, so they had to decrypt the text first (in fact, to solve the problem) for translation. Unfortunately, they were set up twice. First, an original method to crack the cipher suggested by the author didn't really work. It required some improvements. Second, the encrypted text had a 'minor' typo – one line in the middle of the text was missing! Just an annoying printing artifact. But eventually, the Russian editors had solved both the issues and the Russian printing had a few extra chapters of their commentaries and solutions.

Well, back to the original printing in English. Throughout the book the author repeats his main guideline – only by working on real problems can you develop as a programmer. Well, to be honest, it is obvious, but creating or finding such problems is a challenge. This book is a fantastic source of real and interesting problems covering a wide field of computer science, and is peppered with fascinating and inspiring introductions.

Over all these years I've been still trying to solve some etudes from the book, with varying degrees of success, and I'm still fascinated.

A few years ago, I bought an original printing of the book. It was a bit pricy because the book had only one printing back in 1978 and I obtained my copy from Wingate University library, complete with fascinating librarian stamps and markers. I was thrilled.

I hope now you've got yet another book in your check list. ■

ALEXANDER DEMIN

Alexander is a software engineer holding a Ph.D. in Computer Science. He is constantly exploring new technologies and is always ready to drill down into the code with a disassembler to prove that the bug is there. He can be contacted at alexander@demin.ws

Getting One Past The Goalpost

Pete Goodliffe explains why the QA team are your friends.

Fights would not last if only one side was wrong

~ François de la Rochefoucauld

The early 20th century philosophers and purveyors of jaunty tuneful hair, The Beatles, told us all you need is love. They emphasised the point: *love is all you need*. Love; that's it. Literally. Nothing else.

It's incredible how long a career they had given that they didn't need to eat or drink [1].

In our working relationships with other inhabitants of the software factory, we would definitely benefit from more of that sentiment. A little more love might lead to a lot better code! Programming in the Real World is a inter-personal endeavour, and so is inevitably bound up in relationship issues, politics, and friction from our development processes.

We work closely with many people. Sometimes in stressful scenarios.

It is not healthy for our working relationships, nor for the consequent quality of our software, if our teams are not working smoothly together. But many teams suffer these kinds of problem.

As a tribe of developers, one of our rockier relationships is with the QA enclave; largely because we interact with them very closely, often at the most stressful points in the development process. In the rush to ship software before a deadline, we try to kick the software soccer ball past the testing goal-keepers.

So let's look at that relationship now. We'll see why it's fraught, and why it must not be.

Software development: shovelling manure

In unenlightened workplaces the development process is modelled as a huge pipe: conveying raw materials pumped in the top, through various processes, until perfectly formed software gushes (well, perhaps dribbles) out the end:

- Someone (perhaps a *business analyst* or *product manager*) pours some requirements into the mouth of the pipe.
- They flow through architects and designers, where they turn into specifications and pretty diagrams (or good intentions, and smoke and mirrors).
- This flows through the programmers (where the *real* work gets done, naturally), and turns into executable code.
- Then it flows into QA. Where it hits a blockage as the 'perfectly formed' software magically turns into a non-functioning disaster. These people *break* the code!
- *Eventually* the developers push hard enough down the pipe to break this blockage, and the software finally flows out of the far end of the pipe.

In the fouler development environments, this pipe resembles more a sewer. QA feel like the developers are pumping raw sewage down to them, rather than handing them a thoughtfully gift-wrapped present. They feel they are being dumped on, rather than worked with.

Is software development really this linear? Do our processes really work like this simple pipeline (regardless of how pure the contents)?

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

No. They don't.

The pipe is an interesting first approximation (after all, you cant test code that hasn't been written yet), but far too simplistic a model for real development. The linear pipeline view is a logical corollary of our industry's long fascination with the flawed *waterfall* development methodology.

> It is wrong to view software development as a linear process.

However, this view of the development process does explain why the development team's interaction with the QA team isn't as smooth as it should be. Our processes and models of interaction are too often shaped by the flawed sewage-based development metaphor. We should be in constant communication, rather that just throwing them software towards the end of the development effort.

What is QA good for?

To some it's obvious what they do. To others it's a mystery. The 'QA' department (that is, Quality Assurance) exist to ensure that your project ships a software product of sufficient quality. They are a necessary and vital part of the construction process.

What does this entail? The most obvious and practical answer is: they have to test the living daylights out of whatever the developers create in order to ensure:

- that it matches the specification and requirements – that every feature that should be implemented has been implemented,
- that the software works correctly on all platforms – that is, it works on all OSes, on all versions of those OSes, on all hardware platforms, and on all supported configurations (e.g. meeting minimum memory requirements, minimum processor speeds, network bandwidth, etc), and
- that no faults have been introduced in the latest build – the new features don't break any other behaviour, and no regressions (the reintroduction of previous bad behaviour) have been introduced.

Their name is 'QA', not just 'the testing department', and for a reason. Their role is not just pushing buttons like robots; it's baking quality into the product.

To do this QA must be deeply involved throughout, not just a final adjunct to the development process.

- They have a hand in the specification of the software, to understand – and shape – what will be built.
- They contribute to design and construction, to ensure that what's built will be testable.
- They are involved heavily in the testing phase, naturally.
- And also in the final physical release: they ensure that what was tested is what is actually released and deployed.

A false dichotomy

Our inter-team interactions are hindered because they are just that: interactions between *separate* teams. QA people are considered a different tribe, distinct from the 'important' developers. This bogus, partitioned vision of the development organisation inevitably leads to problems.

When QA and development are seen as separate steps, as separate activities, and therefore as very separate teams, an artificial rivalry and disconnect can too easily grow. This is reinforced physically by our building artificial silos between testers and developers. For example:

- The two teams have different managers, and different reporting lines of responsibility.
- The teams are not co-located, and have very different desk locations (I've seem QA in separate desk clusters, on different floors, in different buildings, and even – in an extremely silly case – on another continent).
- There are different team structures, recruiting policies, and expected turnover of staff. Developers are valued resources, whereas testers are seen as replaceable cheap mercenaries.
- And most pernicious: the teams have very different incentives to complete task. For example: the developers are working with the promise of bonus pay if they complete a job quickly, but the testers are not. In this case, the developers rush to write the code (probably *badly*, since they're hurrying). They then get very cross when the QA guys won't sanction it for a timely release.

We reinforce this chasm with stereotypes: developers *create*, testers *break*.

There *is* an element of truth there. There are different activities and different skills required in both camps. But they are not logically separate, silo-ed activities. Testers don't find software faults to just break things and cause merry hell for developers. They do it to improve the final product.

They are there to bake in quality. That's the Q in QA. And they can only do this effectively in tandem with developers.

> Beware of fostering an artificial separation between development and testing efforts.

By separating these activities we breed rivalry and discord. Often development processes pit QA as the *bad guy* against the developer *hero*. Testers are pictured standing in the doorway, blocking a plucky software release getting out. It's as if they are *unreasonably* finding faults in our software. They are nit-picking over minutiae.

It's almost as if they're running into the forests, catching wild bugs, and then injecting them into our otherwise perfect code.

Does that sound silly?

Of course it does when you read it here, but it's easy to start thinking that way:The code is fine; those guys just don't know how to use it. Or: They've been working far too long to find such a basic bug; they really don't know what they're doing.

Software development is not a battle. (Well, it shouldn't be.) We're all on the same side.

Fix the team to fix the code

Conway's famous law describes how team structure dictates software structure [2]. It is popularly paraphrased as 'if you have fours teams writing a compiler, it'll be a four-pass compiler'. Experience shows this to be pretty accurate. In the same way that team *structure* affects the code, so does the *health* of the interactions within the software.

> Unhealthy team interactions result in unhealthy code.

We can improve the quality of our software, and the likelihood of producing a great release, by addressing these health issues: by improving the relationship between developers and QA. Working *together* rather than waging war. *Love*, remember, *is all you need*.

This comes down to the way we interact and work with QA. We should not treat them as puppets whose strings we pull, or to whom we throw and ropey software to test. Instead, we treat them as co-workers. Developers *must* have good rapport with QA: a friendship and camaraderie.

Next time

In the next instalment, we'll look at the practical ways we can work better with these inhabitants of the QA kingdom. We'll look at the major ways that developers interact with QA, and how we can improve. ■

Questions

1. How linearly do you view your software development pipeline? Does each stage flow down in a tidy waterfall, or do you iterate through staged software revisions?
2. How separate is the testing/QA team from you as a developer? Are they closely integrated with the development effort, or held off at arms length? Is this appropriate?
3. How close do you think your working relationship with your QA colleagues is? Should it be better? If so, what steps can *you* take to improve it?
4. What is the biggest impediment to software quality in your development organisation? What is required to fix this?

Notes

[1] Well, Shakespeare's Orsino *did* claim that music was the food of love, so perhaps this is possible.

[2] Conway, Melvin E. (April 1986), 'How do Committees Invest?' *Datamation* 14(5):28–31

Effect of Risk Attitudes on Recall of Assignment Statements (Part 2)

Derek Jones concludes his report of the ACCU 2011 Conference Developer Experiments.

This is the second of a two part article describing an experiment carried out during the 2011 ACCU conference. The first part was published in the previous issue of C Vu.[1] This second part discusses the remember/recall assignment statement component of the experiment; see part 1 for a discussion of the experimental setup. Coding guidelines sometimes recommend that words rather than non-words be used in identifiers and sometimes recommend that prefixes be added to identifiers to denote some property (e.g., E_ to denote a member of an enumerated type). The identifiers used in the assignment problem had letter sequences designed to investigate subject performance differences caused by these two factors.

The format of the task subjects' were asked to perform in this part of the experiment is identical to the memory for assignment statements portion of the experiment performed at the 2006 ACCU conference,[2] with the one difference that four assignment statements rather than three were used as the information to be remembered. See the write-up of that experiment for more details that are omitted here.

One of the results from the 2006 experiment (and earlier experiments measuring memory performance) was that subjects did not make enough mistakes to find any statistically significant correlations between identifier attributes and recall performance. It was hoped that increasing the number of assignment statements from three to four would increase the load on short term memory and result in more mistakes being made.

Characteristics of human memory

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

The following are some of the factors that studies have been found to effect subject recall performance of recently seen lists of information. These factors are expected to have some impact on subject performance in this experiment.

- People pay particular attention to the initial part of a word[3, 4] (this enables them to start looking up a word in the mental lexicon while its remaining sounds are being heard).
- A decrease in word list recall performance for similar sounding words.[5, 6] It is believed that the similarity causes confusion between the various word sound sequences and a subsequent failure to correctly retrieve the original information.
- The extent to which the information to be remembered is already stored in longer term memory subsystems (i.e., known letter sequences such as words).
- The time delay between seeing the information and having to recall it (because the remembered information degrades over time),
- A capacity limit on the total amount of information that can be remembered and shortly afterwards recalled or recognized,

The above findings suggest that subject performance could exhibit one or more of the following patterns (some of which drive performance in different directions):

- performance will be better on problems where the identifiers have different initial letters, compared to problems where the initial identifier letters are the same.

  This could occur both because people have been found to pay more attention to the start of a word and if subjects attempt to optimise performance by remembering just the first letter there is often a STM capacity advantage; while both the spoken form of single letters are represented by a single syllable (except *w* which contains two) and each of the letter sequences used was pronounceable as a single syllable the sound duration of the word syllable is longer.
- performance will be worse on problems where the identifiers have similar spoken forms, compared to problems where the identifier have dissimilar spoken forms,
- performance will be better on problems where the identifiers are known words, compared to problems where the identifiers are non-words.

Source code identifiers contain a variety of different kinds of character sequences. Some are recognizable words or phrases, some abbreviated forms of words or phrases, while others have no obvious association with any language known tot he reader (e.g., they may be acronyms that are unknown to the reader). Reading involves converting these character sequences to sounds and it is to be expected that subjects' memories of an identifier will be sound based, rather than vision based.

Part of the problem subjects are asked to answer involved indicating the identifier that did not appear in a previously seen list of assignment statements. This is a recognition problem, while remembering the value assigned in a recall problem. Studies have found that recognition and recall memory have different characteristics.[7]

The assignment problems

The problems and associated page layout were automatically generated using a C program and various awk scripts to generate troff, which in turn generated postscript. The identifier and constant used in each assignment statement was randomly chosen from the appropriate set and the order of the assignment statements (for each problem) was also randomised. The source code of the C program and scripts is available from the experiments web page.[8]

Due to a fault in the generation script the first 10 problems for each subject all used sets of identifiers where the last two letters of each set of identifiers were the same. The intent was that the randomisation algorithm be applied to the choice of identifiers used for all problems.

Selecting identifiers and integer constants

A sufficient number of letter sequences were created so that most subjects would not encounter the same sequence more than once. In all 50 different words and 50 different non-words were used (see the following list). Given

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

this list the same identifier sequence will repeat after every set of 20 problems seen by a subject. The identifier letter sequences were designed to investigate differences in subject performance caused by two factors:

- all identifiers in the assignment list being words or all pronounceable English non-words (no checks were made for English non-words that were words in other languages and possible known to subjects),
- all identifiers in the assignment list starting with or not starting with the same letter (with all corresponding subsequent letters being the same and so they rhymed).

For simplicity identifiers consisted of a sequence of three letters having the pattern CVC. The following lists the sets of five identifiers used in the assignment problem. Assignment problems were created in groups of 20. Each group of 20 used all of the identifiers appearing in one row. The identifiers used in each assignment problem were selected by randomly choosing a row previously unused for a given subject. Four of the identifiers in the row were randomly selected to be used in the list of the four assignment statements to be remembered. The fifth identifier was used as the *not seen* identifier.

```
cat mat hat pat bat
hen pen men ben yen
hop pop top mop cop
din pin sin kin tin
cub rub tub hub pub
dat lat wat gat tat
gen ren sen ven nen
dop gop vop rop pop
nin rin zin cin lin
fub lub wub mub bub
dig dog dad den dot
lot lip led lap lid
pin pat pod peg pen
sat sir sum sod sad
wag wit won web wig
fot fis fup fep fik
kam kig kus kos kuk
ras rit rus roz ral
tid tol tep tul teb
vib vok vup vek vot
```

The word *pop* accidently appeared in both a word and non-word sequence. The answers to problems where *pop* appeared in a non-word list were not included in the analysis described below.

The non-words have a variety of characteristics, including: the non-word *cin* sounding like the word *sin*, *fot* could be remembered as *foot* + CVC pattern, *roz* rozzer slang for policeman (at least in British English) or abbreviation for the name Rosalyn.

Selecting integer constants

The experimental factor under investigation involves attributes of identifiers and the impact of other kinds of information (mostly involving integer constants) on subject performance needs to be minimized. A good approximation to short term memory requirements is the number of syllables contained in the spoken form of the information. Choosing single digit integer constants containing a single syllable minimises their impact on short term memory load.

The integer constants chosen were 4, 5, 6, 8, and 9 (the digit 7 was not used because its English spoken form has two syllables). To within an order of magnitude they all have the same frequency of occurrence in source code.[9]

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 experimental problems seen by subjects has the same format it is possible that some subjects will detect what they believe to be a pattern in the problems which they then attempt to use to improve their performance.

While the general format of the problem used commonly occurs during program comprehension, the mode of working (i.e., paper and pencil) is rarely used these days; 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., assignment statements) requires pressing keys (or using a mouse) and having located the sought information additional hand movements (i.e., key pressing or mouse movements) are needed to return to the original source location. In this study subjects were only required to tick a box to indicate that they *would refer back* to locate the information. The cognitive effort needed to tick a box is probably a lot less than would be needed to actually refer back. Studies have found[10] 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.

Each identifier appeared once per set of 100 assignment statements. Based on expected subject performance, it was anticipated that most identifiers would be seen once, with only a few identifiers being seem twice by the faster subjects. Thus any learning of individual problem identifier sets by the faster subjects was not expected to have a significant impact on the results.

While subjects were told they are not in a race and that they should work at the rate at which they would normally process code, it is possible that some subjects ignored this request. A consequence of this is that the distribution in the number of problems answered, and perhaps the accuracy of the results, may be different than would occur if all subjects followed the instructions given to them.

If subjects randomly guess answers to questions that they cannot recall answers to, then (given that only five possible numeric values were used and no value occurred more than once in the same problem):

- if a subject knew no answers and randomly guessed the four answers, then an average of 0.88 of a question would be guessed correctly (total 0.88 correct per problem),
- if a subject knew one answer and randomly guessed the other three answers, then an average of 0.875 of a question would be guessed correctly (plus one known answered correctly; total 1.875 correct per problem),
- if a subject knew two answers and randomly guessed the other two answers, then an average of 0.67 of a question would be guessed correctly (plus two known answered correctly; total 2.67 correct per problem),
- if a subject knew three answers and randomly guessed the other one answers, then an average of 0.5 of a question would be guessed correctly (plus three known answered correctly; total 3.5 correct per problem).

Ecological validity

For the results of this experiment to be applicable to professional developer performance it is important that subjects work through problems at a rate similar to that which they would process source code in a work environment. Subjects were told that they are not in a race and that they should work at the rate at which they would normally process code. Experience from previous experiments has shown that the competitive instinct in some developers causes them to ignore the work rate instruction and attempt to answer all of the problems in the time available. To deter such behaviour during this experiment the problem pack contained significantly more problems (28 in total) than subjects were likely to be able to answer in the available time (one subject answered all problems and two subjects all but one).

The structure of the problem follows a pattern that is often encountered when trying to comprehend source code: see information (and try to

remember some of it), perform some other task and then perform a task that requires making use of the previously seen information.

Considering the experimental context as a whole, the constant repetition of exactly the same kind of activity rarely occurs in program development. The constant repetition provides an opportunity for learning to occur, e.g., subjects have the opportunity to tune their performance for a particular kind of problem. The issue of learning and problem solving strategies used by subjects is discussed below.

Results

It was estimated that each subject (20 expected, on the day 30) would be able to answer 20 problem sets (on the day 20.0, sd= 7.7) in 20-30 minutes (on the day 20 minutes). Based on these estimates the experiment would produce 2000 (on the day 2980) individual answers. Table 1 gives a summary of the kinds of the results.

Table 1

Summary of results for this and the 2006 experiment. The 'Total' column is summed over all answers while the 'By subject' column gives the subject mean (standard deviation in brackets) values. The 'Correct recalls', 'Incorrect recalls' and 'Would refer back' percentages are calculated using the total number of answers in those three cases. The 'Not seen (incorrect)' values are calculated using the total number of the *not seen* answers.

	Total	By subject	2006 Total	2006 By subject
Correct recalls	1379 (57.2%)	59.5% (29)	685 (52.2%)	60.8% (26)
Incorrect recalls	397 (16.5%)	15.5% (15)	122 (10.6%)	9.0% (9)
Would refer back	634 (26.3%)	25.0% (28)	349 (30.2%)	30.2% (27)
Not seen (incorrect)	54 (9.5%)	8.4% (10)	20 (5.5%)	5.1% (8)

The average amount of time taken to answer a complete problem was 60 seconds. The format of the experiment means that no information is available on the amount of time invested in trying to remember information, answering the questionnaire sub-problem, and then thinking about the answer to the recall sub-problem (i.e., the effort break down for individual components of the problem). While STM recall performance drops very quickly after the information is no longer visible (studies have found below 10% correct within around 8 seconds in many situations[5]). Even the fastest subject took over 25 seconds per complete problem and so recency effects are likely to be minimal.

Subject strategies

Feedback from subjects who took part in the previous experiments highlighted the use of a variety of strategies to remember information for each assignment problem. The analysis of the threats to validity in some of those experiments has discussed the question of whether subjects traded off effort on the filler task in order to perform better on the assignment problem, or carried out some other conscious combination of effort allocation between the subproblems. To learn about strategies used during this experiment, after 'time' was called on problem answering, subjects were asked to list any strategies they had used (a sheet inside the back page of the handout had been formatted for this purpose).

The responses to the strategies question generally contained a few sentences. Only a few responses involved the questionnaire part of the problem, with subjects saying they answered honestly.

The strategies listed consisted of a variety of the techniques people often use for remembering lists of names or numbers. For instance, number word associations, continuous repetition of information, creating memorable sentences merging words, reordering the sequence presented into a regular pattern (e.g., alphabetical), inventing short stories involving the words and numbers and associating the information with musical sound patterns. One subject gave remembering the first letter as a strategy.

From the replies given it was not possible to work out if subjects give equal weight to answering both parts of the problem, or had a preference to answering one part of the problem.

No subject listed a strategy that was based on the visual appearance of the identifiers or numbers, although several subjects said they tried to associate images with the identifiers and numbers.

Recall/would refer back performance

This subsection treats the value recall and *would refer back* answers as a single subproblem for the purpose of analysis. The not seen answers are treated as a different subproblem and is discussed in the following subsection.

Figure 1 is a scatter plot of the percentage of correct/incorrect recall and *would refer back* answers given by subjects for each of the four kinds of identifiers. The straight line is the set of points along which the values on the two axis sum to 100%.

The ideal subject behaviour is for all points in the correct recall vs. *would refer back* scatter plot to lie along the straight line, i.e., subjects would either give the correct answer or they *would refer back*; see left of Figure 1.

The scatter plot of correct vs. incorrect recall answers shows some points along the 100% line, i.e., in some cases subjects were either right or wrong and never gave a *would refer back* answer (perhaps these subjects are willing to take more risks); see middle of Figure 1.

The scatter plot of incorrect recall against *would refer back* answers shows some points along the 100% line, i.e., in some cases subjects either give incorrect or *would refer back* answers (perhaps these subjects are willing to take more risks); see right of Figure 1.

Self-knowledge, metacognition, is something that enables a person to evaluate the accuracy of the memories they have. Subjects who give many incorrect answers do not accurately evaluate the state of their own memories of previously seen information (i.e., they overestimated the accuracy of their memories). It is also possible that subjects who gave many would refer back answers also showed poor metacognitive performance (i.e., they underestimated the accuracy of their memories and would have mostly given correct answers had they risked a numeric answer). However, it is not possible to evaluate this possibility based on the available data.

Not seen performance

This subsection treats the *not seen* answers as a single subproblem for analyse. There were 570 *not seen* answers of which 9.5% were incorrect. Subjects who randomly chose an answer would achieve a 80% failure rate.

Figure 2 is a scatter plot of the percentage of correct/incorrect recall and *not seen* answers given by subjects for each of the four kinds of identifiers. The results are dominated by the high percentage of correct answers.

Effects of different identifier naming patterns

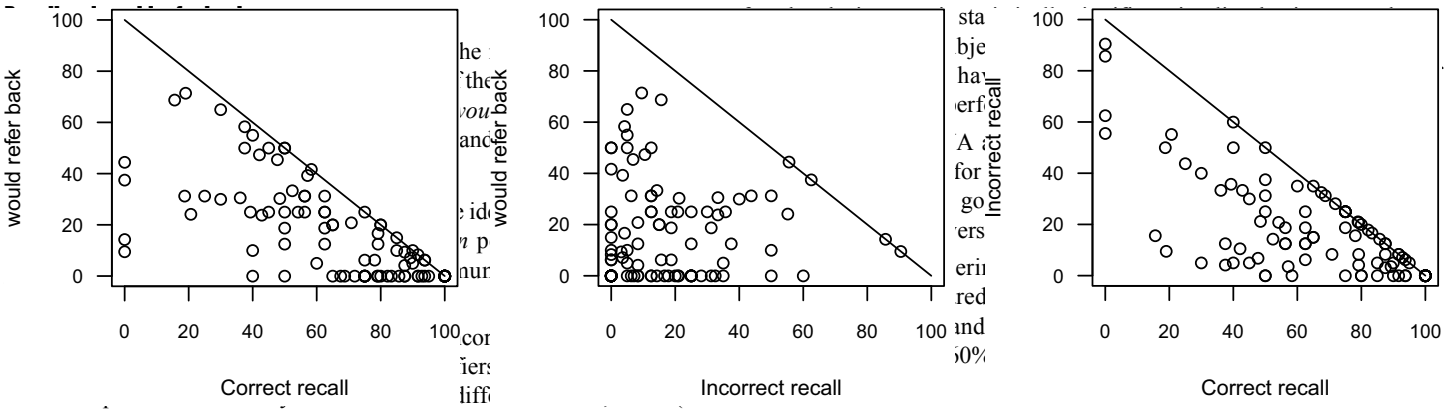
This experiment has a two factorial design with two levels. The factors are 'is word' and 'same first' and the levels are TRUE/FALSE. All four combinations of identifiers were used, enabling the interaction between them to be investigated.

The statistical technique used to analyse the results is ANOVA (analysis of variance). For details of the analysis see the source code of the R program used to analyse the data available for download[8] along with the (anonymous) data extracted from subject answers.

The subject data was split into two groups of response variables, with 'is word' and 'same first' used as the predictor variables in both cases. One group of response variables was percentage of recall correctness and percentage of *would refer back* answers and the other response variable was percentage of incorrect *not seen* answers.

For some combinations of identifier attributes some subjects gave a small number of answers. Answer counts were converted to percentages and to prevent a small number of answers potentially giving a nonrepresentative value any subject data set containing less than five (recall + *would refer back*) answers or less than three *not seen* answers were excluded from the analysis.

Figure 1



Scatter plots of various combinations of percentage correct recall, incorrect recall and would refer back answers plotted against each other. The straight lines are the set of points along which the values on the two axis sum to 100%.

6.8%) and 2.6% (sd 4.5%) error rates respectively. This was difference statistically significant.

Analysis of the data found that differences in the identifier attribute 'is word' was not a good predictor of *not seen* subject performance; ANOVA p-value=0.90. The interaction between the two predictor variables had a p-value of 0.62.

The mean subject percentage of incorrect *not seen* answers was slightly higher when all assignment identifiers contained words compared to all non-words (4.8% vs. 3.6%, with sd of 7.3% and 5.5% respectively), but this difference was not statistically significant.

Comparison of 2011 results with 2006

How do the results of the 2006 and 2011 experiment compare? Both ran for 20 minutes and subjects completed subjects completed almost the same number of problems.

The summary in Table 1 shows that recall and *would refer back* percentages are similar. There is a higher percentage of incorrect recalls in 2011, as might be expected with the increased amount of information that has to be remembered.

An analysis of the 2011 and 2006 subject answers finds a statistically significant difference in the mean percentage of 'incorrect recall' answers (Student's t-test gives p-value=0.012; the 95% confidence interval for the 2011 mean percentage is between 1.3 and 10.1 greater than the 2006 value) but not for 'correct recall' or *would refer back* (p-values of 0.55 and 0.30 respectively).

If subjects' percentage of incorrect recall answers increases, the percentage for correct recall and/or *would refer back* must decrease. The filler problems used in the 2006/2011 experiments both involved providing answer to a short list if questions, i.e., making use of existing knowledge to solve a problem that only required a small amount of information to be held in STM.

Conclusion

The 2011 results suggest that when developers have to recall information about a recently seen list of identifiers they make more misidentification mistakes when those identifiers all start with the same letter compared to when they start with different letters; the 2006 results do not replicate this finding. If authors of coding guidelines practice feel it is worthwhile adding a fix sequence of letters to an identifier to denote some attribute, the number of mistakes made when reading these identifiers might be reduced if these characters were added somewhere other than at the start of the identifier (e.g., at the end).

No significant performance difference was found between assignment lists using identifiers that were all words or all non-words.

The results were consistent with the finding of two previous experiments where the breakdown of subjects answers was approximately: *would refer back* 25%, recall correct 60% and recall incorrect 15% (see Table 1). Increasing the number of to-be remembered assignment statements from three to four results the number of incorrect recall answers increasing.

Future experiments might confirm and extend these findings to identifiers containing more than three letters and investigate subject performance when handling identifiers having other attributes. ■

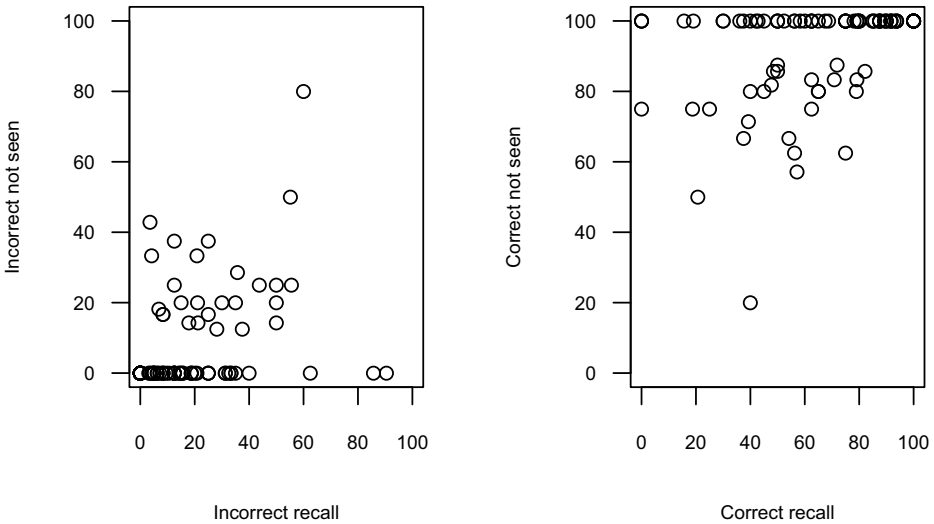
Further reading

Statistics Explained by Perry R. Hinton provides a very good introduction to statistics, including ANOVA. 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 and those involved in organising the ACCU conference for making a conference slot available in which to run it.

Figure 2



Scatter plots of various combinations of percentage correct recall, incorrect recall and would refer back answers plotted against each other. The straight lines are the set of points along which the values on the two axis sum to 100%.

Using D Slices

Steven Schveighoffer looks at slices in D and shows why they're not arrays.

One of the most pleasant features of the D language is its implementation of slices. Every time I use a programming language that isn't D, I find myself lamenting for D's slice syntax. Not only is it concise and efficient, but things 'just work' when you are dealing with slices.

I'll go over some of the background and internals of D slices and arrays, and hopefully after reading this, you will have a clearer understanding of the proper ways to use D slices, as well as an idea of how fundamentally different they are than normal arrays!

An overflowing problem

In most languages, an array is a built-in type which manages its own data, and is passed around by reference. One refers to the entire thing as an 'Array', and associates all the operations for the array (such as setting values, appending data for dynamic arrays, obtaining the length) to that type.

However, D takes its lineage from C, where an array is simply a chunk of contiguous data. In C, a reference to an array or array segment is as simple as a pointer (an explicit reference). C's arrays are distinctly unmanaged by the type that refers to them – the pointer. The only operations supported are to retrieve and set data using an offset from the pointer.

I know most of you are probably familiar with array syntax in C, but there are some languages out there which use different syntax. So for their benefit, here are some examples of array usage in C:

```
arr[0] = 4; /* sets the first element
               of the array 'arr' to 4 */
x = arr[1]; /* retrieves the second element
               of the array 'arr' into x */
```

Everything else (length, appending, allocation, destruction) is left up to library functions and assumption/documentation. So what is so wrong with this concept? One of the largest problems with C arrays is the ability to access any data via the pointer, even data that doesn't belong to the array. You can even use negative indexes! Not to mention that the array uses the exact same type as a pointer to a single item. When you get a pointer as a parameter to a function, that could be an array, or it could be just a pointer to a single item. Cue buffer overflow attacks. You can read more about this in Walter Bright's article[1].

STEVEN SCHVEIGHOFFER

Steven graduated with a bachelor's degree in CS from Worcester Polytechnic Institute. He's worked as a C++, C#, and web developer for the last 12 years, has largely rewritten the array runtime library for D, and has his own D side project, dcollections.

Effect of Risk Attitudes on Recall of Assignment Statements (continued)

References

[1] D. M. Jones. 'Effects of risk attitude on recall of assignment statements' *C Vu*, 23(6):19–22, Jan. 2012.

[2] D. M. Jones. 'Developer beliefs about binary operator precedence' *C Vu*, 18(4):14–21, Aug. 2006.

[3] H.-F. Chitiri and D. M. Willows. 'Word recognition in two languages and orthographies: English and Greek' *Memory & Cognition*, 22(3):313–325, 1994.

[4] M. Taft and K. I. Foster. 'Lexical storage and retrieval of polymorphemic and polysyllabic words' *Journal of Verbal Learning and Verbal Behavior*, 15:607–620, 1976.

[5] A. D. Baddeley. 'How does acoustic similarity influence short-term memory?' *Quarterly Journal of Experimental Psychology*, 20:249–264, 1968.

[6] V. Coltheart. 'Effects of phonological similarity and concurrent irrelevant articulation on short-term-memory recall of repeated and novel word lists' *Memory & Cognition*, 21(4):539–545, 1993.

[7] J. R. Anderson. *Learning and Memory: An Integrated Approach* John Wiley & Sons, Inc, second edition, 2000.

[8] D. M. Jones. Experimental data and scripts for effects of risk attitude on recall of assignment statements study. http://www.knosof.co.uk/dev-experiment/accu11.html, 2011

[9] D. M. Jones. *The new C Standard: An economic and cultural commentary* Knowledge Software, Ltd, 2005.

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

Listing 1

```
import std.stdio;
void main()
{
  int[] a; // a is a slice

  a = new int[5]; // allocate a dynamic array of
                  // integers that has at least 5
                  // elements, and give me a
                  // slice to the first 5. Note
                  // that all data in D is
                  // default assigned, int's are
                  // defaulted to 0, so this
                  // array contains five 0's

  int[] b = a[0..2]; // This is a 'slicing'
                     // operation.  b now refers
                     // to the first two elements
                     // of a. Note that D uses
                     // open interval for the
                     // upper limit, so a[2] is
                     // not included in b.

  int[] c = a[$-2..$]; // c refers to the last
                       // two elements of a ($
                       // stands for length
                       // inside a slice or index
                       // operation).

  c[0] = 4;  // this also assigns a[3]
  c[1] = 5;  // this also assigns a[4]

  b[] = c[]; // assign the first two elements of
             // a[] to the value from the last
             // two elements (4, 5).

  writeln(a);  // prints "[4, 5, 0, 4, 5]"

  int[5] d;  // d is a fixed sized array,
             //allocated on the stack
             // b = d[0..2]; slices can point at
             // fixed sized arrays too!
}
```

Introducing slices

So how does D improve things? In many ways, D's arrays are similar to C's arrays. In fact, D supports C's array syntax using pointers. However, D provides a new type that builds on C array syntax called a slice. A slice is a segment of an array (dynamic or otherwise) that tracks both the pointer and the length of the segment. With the combined protection of having the length of the data, and the garbage collector to manage the memory backing the data, slices are an extremely powerful, dynamic concept that is safe from most memory corruption issues. In addition, D slices support extending with simple functions which take a slice as the first parameter. This allows one to add any functionality you want to a built-in type via properties or methods. With D slices, one can write high-performance code with elegant and concise syntax that is awkward or inefficient in almost any other language.

So let's see some D slices in action (Listing 1).

You may notice something puzzling about the description of the allocation of the array: 'allocate a dynamic array of integers that has at least 5 elements, and give me a slice to the first 5'. Why isn't it just 'allocate a dynamic array of 5 elements'? Even experienced D coders have trouble with D's array concepts sometimes, and for quite good reason. D's slices are *not* proper dynamic array types (at least not under the hood) even though they appear to be. What they do is provide a safe and easy interface to arrays of any type (dynamic or otherwise). So let's discuss probably the most common misconception of D slices.

Listing 2

```
import std.stdio;

void shrinkTo2(int[] arr)
{
  if(arr.length > 2)
    arr.length = 2;
}

void main()
{
  int[] arr = new int[5];
  arr.shrinkTo2(); // note the ability to call
                   // shrinkTo2 as a method
  writeln(arr.length); // outputs 5
}
```

Who's responsible?

A slice in D seems like a dynamic array in almost all aspects of the concept – when passed without adornments, the data referred to is passed by reference, and it supports all the properties and functions one would expect a dynamic array type to support. But there is one very important difference. A slice does not *own* the array, it *references* the array. That is, the slice is not responsible for allocation or deallocation of its data. The responsible party for managing a dynamic array's memory is the D runtime.

So where is the true dynamic array type in D? It's hidden by the runtime, and in fact, has no formal type. Slices are good enough, and as it turns out, the runtime is smart enough about what you want to do with the data, that you almost never notice dynamic arrays are missing as a full-fledged type. In fact, most D coders consider the D slice to be the dynamic array type – it's even listed as a dynamic array type in the spec! The lack of ownership is very subtle and easy to miss.

Another consequence of this is that the length is not an array property, it's a slice property. This means the length field is not necessarily the length of the array, it's the length of the slice. This can be confusing to newcomers to the language. For instance, this code has a large flaw in it (Listing 2).

This might look like you changed the passed **arr**'s length to 2, but it actually did not affect anything (as is proven by the output from **writeln**). This is because even though the data is passed by reference, the actual pointer and length are passed by value. Many languages have an array type whose properties are all passed by reference. Notably, C# and Java arrays are actually fully referenced Objects. C++'s vector either passes both its data and properties by reference or by value.

To fix this problem, you can do one of two things. Either you explicitly pass the slice by reference via the **ref** keyword, or you return the resulting slice to be reassigned. For example, here is how the signature would look if the slice is passed by reference:

```
void shrinkTo2(ref int[] arr)
```

Let's say you make this change, what happens to the data beyond the second element? In D, since slices don't own the data, it's still there, managed by the nebulous dynamic array type. The reason is fundamental: some other slice may still be referencing that data! The fact that no single slice is the true owner of the data means no single slice can make any assumptions about what else references the array data.

What happens when no slices reference that data? Enter D's garbage collector. The garbage collector is responsible for cleaning up dynamic arrays that no longer are referenced by any slices. In fact, it is the garbage collector that makes much of D's slice usage possible. You can slice and serve up segments of dynamic arrays, and never have to worry that you are leaking memory, clobbering other slices, or worry about managing the lifetime of the array.

A slice you can append on

D's slices support appending more data to the end of the slice, much like a true dynamic array type. The language has a specific operator used for

Listing 3

```
int[] a;  // an empty slice references no data,
          // but still can be appended to
a ~= 1;   // append some integers (this
          // automatically allocates a new
a ~= 2;   // array to hold the elements).

a ~= [3, 4]; // append another array (this time,
             // an array literal)
a = a ~ a;   // concatenate a with itself, a is
             // now [1, 2, 3, 4, 1, 2, 3, 4]

int[5] b;    // a fixed-size array on the stack
a = b[1..$]; // a is a slice of b
a ~= 5;      // since a was pointing to stack
             // data, appending always
             // reallocates, but works!
```

concatenation and appending, the tilde (**~**). Listing 3 contains some operations that append and concatenate arrays.

Anyone who cares about performance will wonder what happens when you append the four elements. The slice does not own its data, so how does one avoid reallocating a new array on each append operation? One of the main requirements of D slices are that they are efficient. Otherwise, coders would not use them. D has solved this problem in a way that is virtually transparent to the programmer, and this is one of the reasons slices seem more like true dynamic arrays.

How it works

Remember before when we allocated a new array, I said *allocate a dynamic array of **at least** n elements and give me a slice*? Here is where the runtime earns its keep. The allocator only allocates blocks in powers of 2 up to a page of data (in 32-bit x86, a page of data is 4096 bytes), or in multiples of pages. So when you allocate an array, you can easily get a block that's larger than requested. For instance, allocating a block of five 32 bit integers (which consumes 20 bytes) provides you a block of 32 bytes. This leaves space for 3 more integers.

It's clearly possible to append more integers into the array without reallocating, but the trick is to prevent 'stomping' on data that is valid and in use. Remember, the slice doesn't know what other slices refer to that data, or really where it is in the array (it could be a segment at the beginning or the middle). To make the whole thing work, the runtime stores the number of used bytes inside the block itself (a minor drawback is that the usable space in the block is not as big as it could be. In our example, for instance, we can truly only store 7 integers before needing to reallocate into another block).

When we request the runtime to append to a slice, it first checks to see that both the block is appendable (which means the **used** field is valid), and the slice *ends* at the same point valid data ends (it is not important where the slice begins). The runtime then checks to see if the new data will fit into the unused block space. If all of these checks pass, the data is written into the array block, and the stored **used** field is updated to include the new data. If any of these checks fail, a new array block is allocated that will hold the existing and new data, which is then populated with all the data. What happens to the old block? If there were other slices referencing it, it stays in place without being changed. If nothing else is referencing it, it becomes garbage and is reclaimed on the next collection cycle. This allows you to safely reallocate one slice without invalidating any others. This is a huge departure from C/C++, where reallocating an array, or appending to a vector can invalidate other references to that data (pointers or iterators).

The result is an append operation which is not only efficient, but universally handy. Whenever you want to append a slice, you can, without worry about performance or corruption issues. You don't even have to worry about whether a slice's data is heap allocated, stack allocated, in ROM, or even if it's null. The append operation always succeeds (given you have enough memory), and the runtime takes care of all the dirty work in the background.

Listing 4

```
import std.stdio;

char[] fillAs(char[] buf, size_t num)
{
  if(buf.length < num)
    buf.length = num; // increase buffer length
                      // to be able to hold
                      // the A's
  buf[0..num] = 'A';  // assign A to all
                      // the elements
  return buf[0..num]; // return the result
}
```

Determinism

There is one caveat with slice appending that can bite inexperienced, and even experienced D coders: the apparent non-deterministic behaviour of appending.

Let's say we have a function which is passed a buffer, and writes some number of A's to the buffer (appending if necessary), returning the filled buffer (see Listing 4).

What's wrong with the **fillAs** function? Nothing really, but what happens if increasing the length forces the buffer to be reallocated? In that case, the buffer passed in is *not* overwritten with A's, only the reallocated buffer is. This can be surprising if you were expecting to continue to use the same buffer in further operations, or if you expected the original buffer to be filled with A's. The end result, depending on whether the block referenced by **buf[]** can be appended in place, is the caller's slice might be overwritten with A's, or it might not be. (See Listing 5, which continues the example begun in Listing 4.)

If you give this some thought, you should come to the conclusion that such a situation is unavoidable without costly copy-on-append semantics – the system cannot keep track of every slice that references the data, and you have to put the new data somewhere. However, there are a couple of options we have to mitigate the problem:

1. Re-assign the slice to the return value of the function. Note that the most important result of this function is the return value, not whether the buffer was used or not.
2. Don't use the passed in buffer again. If you don't use the source slice again, then you can't experience any issues with it.

As the function author, there are some things we can do to avoid causing these problems. It's important to note that the only time this situation can occur is when the function appends to, or increases the length of, a passed in slice **and then** writes data to the original portion of the slice. Avoiding this specific situation where possible can reduce the perception of non-determinism. Later we will discuss some properties you can use to predict

Listing 5

```
void main()
{
  char[] str = new char[10]; // Note, the block
  //capacity allocated for this is 15 elements

  str[] = 'B';
  fillAs(str, 20); // buffer must be reallocated
                   // (20 > 15)
  writeln(str);    // "BBBBBBBBBB"
  fillAs(str, 12); // buffer can be extended in
                   // place (12 <= 15)!
   writeln(str);   // "AAAAAAAAAA";
}
```

how the runtime will affect your slice. It is a good idea to note in the documentation how the passed in slice might or might not be overwritten.

A final option is to use ref to make sure the source slice is updated. This is sometimes not an option as a slice can easily be an rvalue (input only). However, this does not fix the problem for any aliases to the same data elsewhere.

Caching

One of the issues with appending to a slice is that the operation is quick, but not quick enough. Every time we append, we need to fetch the metadata for the block (its starting address, size, and *used* data). Doing this means an `O(lg(n))` lookup in the GC's memory pool for every append (not to mention acquiring the global GC lock). However, what we want is amortized constant appending. To achieve this lofty goal, we employ a caching technique that is, as far as I know, unique to D.

Since D2 has introduced the concept of default thread local storage, the type system can tell us whether heap data is local to the thread (and most data is), or shared amongst all threads. Using this knowledge, we can achieve lock-free caching of this metadata, with one cache per thread. The cache stores the most recent n lookups of metadata, giving us quick access to whether a slice can be appended.

This cached lock-free lookup has made array appending even faster than D1's append operation, which suffers from the possibility of data stomping.

Slice members and the appender

With D slices having such interesting behaviour, there is a need sometimes to be able to predict the behaviour of slices and appending. To that end, several properties and methods have been added to the slice.

- `size_t reserve(size_t n)`: Reserves *n* elements for appending to a slice. If a slice can already be appended in place, and there is already space in the array for at least *n* elements (*n* represents both existing slice elements and appendable space), nothing is modified. It returns the resulting capacity. See Listing 6.
- `size_t capacity`: A property which gives you the number of elements the slice can hold via appending. If the slice cannot be appended in place, this returns 0. Note that capacity (if non-zero) includes the current slice elements. See Listing 7.
- `assumeSafeAppend()`: This method forces the runtime to assume it is safe to append a slice. Essentially this adjusts the used field of the array to end at the same spot the slice ends. See Listing 8.

If D slices' append performance just isn't up to snuff for your performance requirements, there is another alternative. The `std.array.Appender` type will append data to an array as fast as possible, without any need to look up metadata from the runtime. `Appender` also supports the output range idiom via an append operation (normal slices only support the output range by overwriting their own data). For more information on ranges and the `Appender` type, see the D online documentation[2].

Listing 6

```
int[] slice;
slice.reserve(50);
foreach(int i; 0..50)
   slice ~= i;        // does not reallocate
```

Listing 7

```
int[] slice = new int[5];
assert(slice.capacity == 7); // includes the 5
                     // pre-allocated elements
int[] slice2 = slice;
slice.length = 6;
assert(slice.capacity == 7); // appending in
       // place doesn't change the capacity.
assert(slice2.capacity == 0); // cannot append
               // slice2 because it would stomp
               // on slice's 6th element
```

Listing 8

```
int[] slice = new int[5];
slice = slice[0..2];
assert(slice.capacity == 0); // not safe to
        // append, there is other valid data
        // in the block.
slice.assumeSafeAppend();
assert(slice.capacity == 7); // force the used
        // data to 2 elements
```

Conclusion

Whether you are a seasoned programmer or a novice, the array and slice concepts in D provide an extremely rich feature set that allows for almost anything you would want to do with an array type. With a large focus on performance and usability, the D slice type is one of those things that you don't notice how great it is until you work with another language that doesn't have it. I highly recommend to anyone who is not familiar with D to at least play around with some of the D slice syntax, and discover how straightforward array programming can be. ■

Acknowledgements

Thanks to David Gileadi, Andrej Mitrovic, Jesse Phillips, Alex Dovhal, Johann MacDonagh, and Jonathan Davis for their reviews and suggestions for this article.

References

[1] http://drdobbs.com/blogs/cpp/228701625
[2] http://d-programming-language.org/

Holiday Rules

Omar Bashir provides an implementation of calendars and holiday rules in Java.

Most business applications depend upon calendars. These calendars define various events in a particular calendar year. Many of these events are specific to the business domain. Events that are common across all domains are holidays. Various business tasks require the knowledge of holidays during their processing, for example, financial transactions are settled on business days and not on holidays. In several implementations, these calendars are partitioned based on the types of events. For example, an application may have a separate calendar specifying holidays (i.e., holiday calendar) and a separate calendar specifying production schedules of a plant, etc. While specification of holidays in an application may have little significance as compared to the overall business logic, failure of holiday calendars can cause a much greater nuisance and in certain circumstances, even result in a financial loss.

In their simplest form, calendars are implemented in software applications as lists of events and their corresponding dates stored in databases or files for many years in advance. Most global businesses need to maintain a set of calendars for each theatre of operation. This can easily lead to a considerable amount of calendar data, acquiring and maintaining which can be costly. There are also risks related to errors and omissions in these data and these risks grow substantially with the amount of data.

While this may be necessary for certain calendars, it is possible to generate holiday calendars from a smaller (mostly fixed or rarely changing) set of rules. These rules can be interpreted as required to generate holidays for a given year. If the rules and the code interpreting these rules are comprehensively tested, risks related to errors and omissions can be significantly reduced. Storage and acquisition costs in rule-based calendars are small and mostly constant as compared to maintaining individual dates of holiday calendars.

It may not always be possible to entirely automate a holiday calendar for a specific region. For example, Islamic holidays follow a lunar calendar and are traditionally subject to the sighting of the moon in many countries. Therefore, any implementation of a holiday calendar based on rules must allow exceptions to be specified manually for particular days to be included into as well as excluded from the holiday calendar of a specified year. An interesting example is moving the UK's Spring Bank Holiday in 2012 from 28 May 2012 to 4 June 2012 to create a long weekend for Queen's Diamond Jubilee. While 4 June 2012 and 5 June 2012 have to be added manually as holidays in such a calendar generator, holiday rules for UK will indicate 28 May 2012 as a holiday, which will have to be ignored as an exception.

This article illustrates rules for determining holidays. Instances of these rules are combined into rule sets for determining holidays for a given year. As several variations (e.g., regional, organisational, religious etc.) of holiday rules may exist, it is not possible to describe all the different holiday rules and rule sets. An object oriented implementation in Java of the English holiday calendar using rules to determine holidays for a given year is then described.

OMAR BASHIR

A programmable calculator with 0.5 kB of user memory and a very basic version of BASIC inspired Omar into computing in 1986. His interests include distributed systems and interesting applications of design patterns. He can be contacted at obashir@yahoo.com

Holiday calendar rules

Holidays are of two types, fixed and floating. Fixed holidays occur either on a particular date or a particular day of a particular month every year whether that day is a weekend or another holiday. Floating holidays can fall on different days in different years. A typical example of a floating holiday is the Easter weekend, which falls on different dates every year. Additionally, in some countries, certain holidays may be rolled forward or backward if they coincide with weekend or any other holiday.

Therefore, the simplest holiday rule for fixed holidays would contain three attributes, the day of the month, the month and the year. If values exist for all these attributes in an instance of this rule, the rule interpreter should interpret it as a single holiday on that particular date. If the value for the year is omitted (i.e., the year is null), it should be interpreted as a holiday on the specified day of the month and the specified month every year. Thus, by extension, if only the value for day exists, it should be interpreted as a holiday on the specified day of the month for every month and in every year.

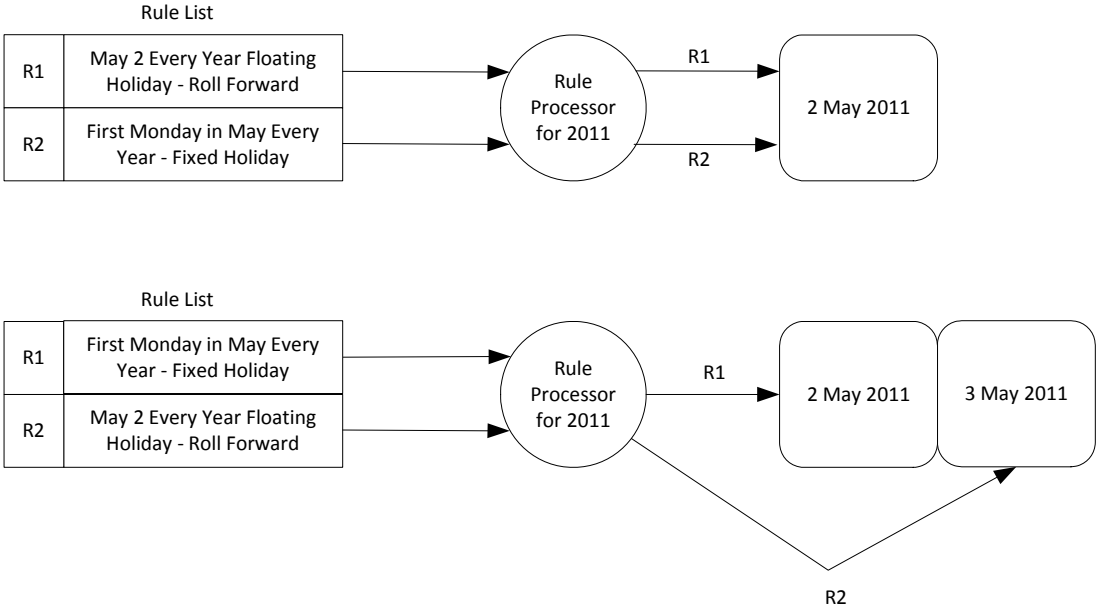
The fixed holiday rule can be extended to a floating holiday rule by including one more parameter. This parameter determines whether the holiday is to be rolled forward, rolled back or not rolled. Rolling is the process of changing the date to the next (roll forward) or the previous (roll back) working day. Thus, the floating holiday rule requires the knowledge to the weekend convention as well as whether the date rolled to is a holiday or not. Weekend convention in most of the world is Saturday and Sunday. A holiday falling on any of these days may have to be rolled forward or back to the next or previous working day.

Certain holidays are dependant on the lunar calendar. These include the Easter. Easter Day is the first Sunday after the full moon that occurs on or after the vernal equinox. This full moon is not the astronomical full moon but the ecclesiastical moon. It is determined using astronomical rules and tables and usually is in synchronisation with the astronomical full moon. According to these rules, the vernal equinox is fixed to 21 March and the ecclesiastical full moon is the 14th day of a tabular lunation (new moon). The Astronomical Applications Department of the US Naval Observatory has produced an algorithm that provides the date of Easter for the given year [1]. An implementation of this algorithm to provide dates for Good Friday and Easter Monday is given in the following section.

Instances of holiday rules are organised in a holiday rule set. Rules in a rule set are interpreted for a given year considering the weekend convention and a list of exceptions. Exceptions are dates that the holiday calendar may determine as holidays but are to be ignored as they have been declared as working days for that particular year for a specific reason. Fixed holiday rules have precedence in interpretation over floating holiday rules. For example, consider that a fixed holiday rule and a floating holiday rule return the same date and the fixed holiday rule is interpreted earlier. Later, when the floating holiday rule is interpreted to get the same date, that date is rolled forward or backward depending on the floating rule thereby correctly resulting in two holidays. On the contrary, if the floating holiday rule is interpreted earlier, only one holiday will exist for both the rules. This is because the fixed holiday, which cannot be rolled, is determined after the floating holiday occurring on the same date. Therefore, rules for fixed holidays should be interpreted before rules for floating holidays.

For instance, a fixed holiday rule states that first Monday of May every year is a fixed holiday. A floating holiday is also specified for 2nd of May

Figure 1



every year to be rolled forward if 2nd May is a holiday. 2 May 2011 is the first Monday of May. If the floating holiday rule is interpreted first, it will return 2 May 2011. Because it has not been determined as a holiday earlier, it is added to the collection of holidays. When the fixed holiday rule is now interpreted, it also returns 2 May 2011 as a holiday. This date is ignored as 2 May 2011 has already been determined as a holiday. However, if the order of interpretation of these rules is revered, the fixed holiday rule returns 2 May 2011 as a holiday first. Next, when the floating holiday rule is interpreted it determines that 2 May 2011 is already a holiday. Because the rule allows rolling the holiday forward, the rule returns 3 May 2011 as a holiday (see Figure 1).

Implementing the English holiday calendar

The UK holiday calendar is relatively simple as compared to many other holiday calendars. It contains eight holidays that are floating. If any of these holidays falls on the weekend or any other holiday, it is rolled forward to the next business day. Rules for these holidays and results of the interpretation of these rules for three years are listed in table 1. The last three rows of the table specify additional exceptional holidays for specific years.

Holiday rules for UK can thus further be classified into the following:

1. Easter Rule to provide dates for Good Friday and Easter Monday.
2. Date Rule to specify holidays as day of month and month. These holidays recur on the specified days of the month for the specified months every year. Optionally the year may be specified for holidays that occur only once. Date Rules can be specified to roll holidays forward to the next business day if they fall on a weekend or any other holiday.
3. Day of Week Rule to specify holidays that occur on a particular day of the week for a given month, for example, 3rd Monday in July or the last Thursday in February. As with the Date Rule, these rules can also be specified to roll holidays forward to the next business day if they fall on a weekend or any other holiday.

Table 2 is a representation of Table 1 using the above mentioned holiday rules for the UK holiday calendar.

In addition to the rules specified in Table 2, a list of exclusion dates must also be included. Once the rule interpreter has determined the holidays for a given year, if one of these holidays exists in the list of exclusion dates, it is omitted from the list of holidays to be returned.

Figure 2 (overleaf) shows an object oriented implementation of these rules based on the STRATEGY design pattern [2].

A class providing functionality of a holiday rule implements the **`HolidayRule`** interface (Listing 1).

The **`interpret`** method is used to interpret the rule. It requires a list containing holidays already determined by interpreting other rules. Holiday that the **`interpret`** method determines is appended to this list. It also requires the year for which this rule is to be interpreted and the list of exclusion dates. If the rule is interpreted to a date contained in the list of exclusion dates, that date is ignored and not added to the list of holidays.

The abstract class **`AbstractHolidayRule`** (Listing 2) implements the **`HolidayRule`** interface. It provides the common functionality for the **`DateRule`** (Listing 3) and **`DayOfWeekRule`** (Listing 4) classes implementing the Date Rule and Day of Week Rule respectively. These three classes employ the TEMPLATE METHOD design pattern [2].

Listing 1

```
public interface HolidayRule {
  void interpret(List<Date> holidays,
                 int calcYear,
                 List<Date> exclusionDates);
  boolean isRollOver();
}
```

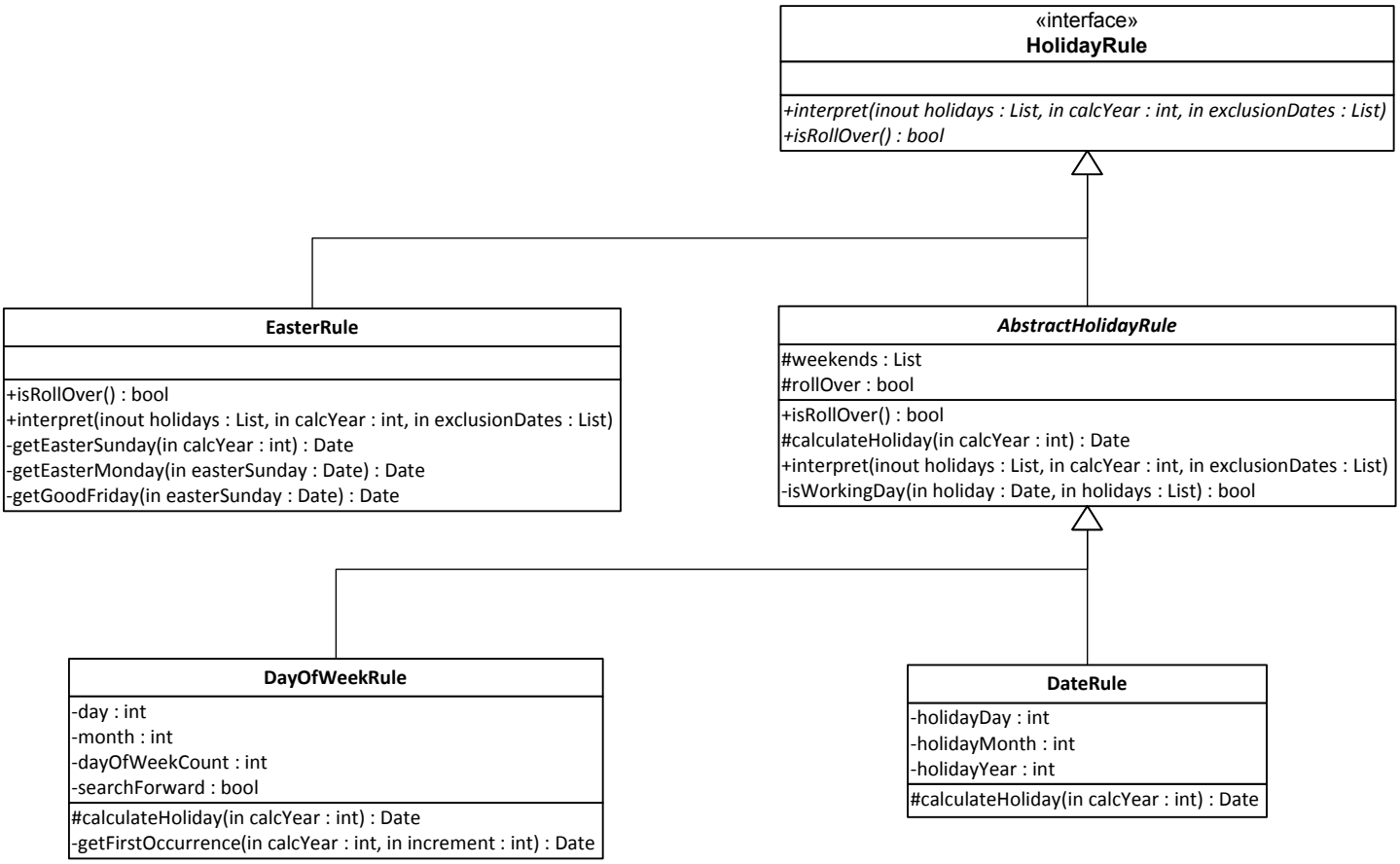
Table 1

Rules	2011	2012	2013
New Years Day, Jan 1	3 Jan	2 Jan	1 Jan
Good Friday	22 Apr	6 Apr	29 Mar
Easter Monday	25 Apr	9 Apr	1 Apr
Early May Holiday, First Monday of May	2 May	7 May	6 May
Spring Holiday, Last Monday of May	30 May		27 May
Summer Holiday, Last Monday of August	29 Aug	27 Aug	26 Aug
Christmas, Dec 25	26 Dec	25 Dec	25 Dec
Boxing Day, Dec 26	27 Dec	26 Dec	26 Dec
Royal Wedding (2011)	29 Apr		
Alternative Spring Break (2012)		4 Jun	
Diamond Jubilee (2012)		5 Jun	

Table 2

Rule	Day of Month	Month	Year	Day of Week	Day of Week Count	Day of Week Direction	Roll
Date	1	1					True
Easter							False
Day of Week		5		Mon	1	Forward	True
Day of Week		5		Mon	1	Reverse	True
Day of Week		8		Mon	1	Reverse	True
Date	25	12					True
Date	26	12					True
Date	29	4	2011				False
Date	4	6	2012				False
Date	5	6	2012				False

Figure 2



Listing 2

```
public abstract class AbstractHolidayRule
   implements HolidayRule {
  protected boolean rollOver;
  private List<Integer> weekends;

  public AbstractHolidayRule(
     List<Integer> weekends, boolean rollOver){
    this.weekends = weekends;
    this.rollOver = rollOver;
  }

  @Override
  public boolean isRollOver(){
    return this.rollOver;
  }

  protected abstract Date calculateHoliday
     (int calcYear);

  @Override
  public void interpret(List<Date> holidays,
     int calcYear, List<Date> exclusionDates) {
    Date holiday =
       this.calculateHoliday(calcYear);
    if (null != holiday){
      if (this.rollOver){
        while (!isWorkingDay(holiday, holidays)){
          holiday.add(Calendar.DAY_OF_MONTH, 1);
        }
      }

      if ((!holidays.contains(holiday)) &&
        (!exclusionDates.contains(holiday))){
        holidays.add(holiday);
      }
    }
  }
```

Listing 2 (cont'd)

```
  private boolean isWorkingDay(Date holiday,
     List<Date> holidays) {
    boolean reply = true;
    if (holidays.contains(holiday)){
      reply = false;
    else {
      if (this.weekends.contains
         (holiday.getDayOfWeek())){
        reply = false;
      }
    }
    return reply;
  }
}
```

The `interpret` method in the `AbstractHolidayRule` calls a protected abstract method `calculateHoliday` which is implemented in the two subclasses of `AbstractHolidayRule` class. This method takes as argument the year for which the holiday is to be determined. If the holiday returned is not a working day, i.e., it is a weekend or it already exists in the list of holidays populated by interpreting preceding rules, and the `rollOver` flag is set, it rolls the day over to the next working day. If the `rollOver` flag is not set, then the date returned by the `calculateHoliday` method is accepted. Finally, this date is added to the list of holidays if it does not already exist in that list.

Implementation of `calculateHoliday` in the `DateRule` class creates the an object of the `Date` class with its `month`, `day` and `year` attributes set to the values of the respective attributes in the object of the `DateRule` class if the `holidayYear` attribute in that object of the `DateRule` class is not null and is equal to the `calcYear` argument to the method. If the `holidayYear` attribute in the object of the `DateRule` class is null, then an object of the `Date` class is returned with the `month` and the `day` attribute values set to the `holidayMonth` and `holidayDay` attribute values of the object of the `DateRule` class and the year attribute set to the `calcYear` argument to the method.

Listing 3

```
public class DateRule extends AbstractHolidayRule
{
  private Integer holidayDay;
  private Integer holidayMonth;
  private Integer holidayYear;
  public DateRule(List<Integer> weekends,
     Integer holidayDay, Integer holidayMonth,
     Integer holidayYear, boolean rollOver) {
    super(weekends, rollOver);
    this.holidayDay = holidayDay;
    this.holidayMonth = holidayMonth;
    this.holidayYear = holidayYear;
  }
  @Override
  protected Date calculateHoliday(int calcYear) {
    Date date = null;
    if (null != holidayYear){
      if (calcYear ==
         this.holidayYear.intValue()){
        date =
          new Date(this.holidayYear.intValue(),
                   this.holidayMonth.intValue(),
                    this.holidayDay.intValue());
      }
    } else {
      date = new Date(calcYear,
                   this.holidayMonth.intValue(),
                    this.holidayDay.intValue());
    }
    return date;
  }
}
```

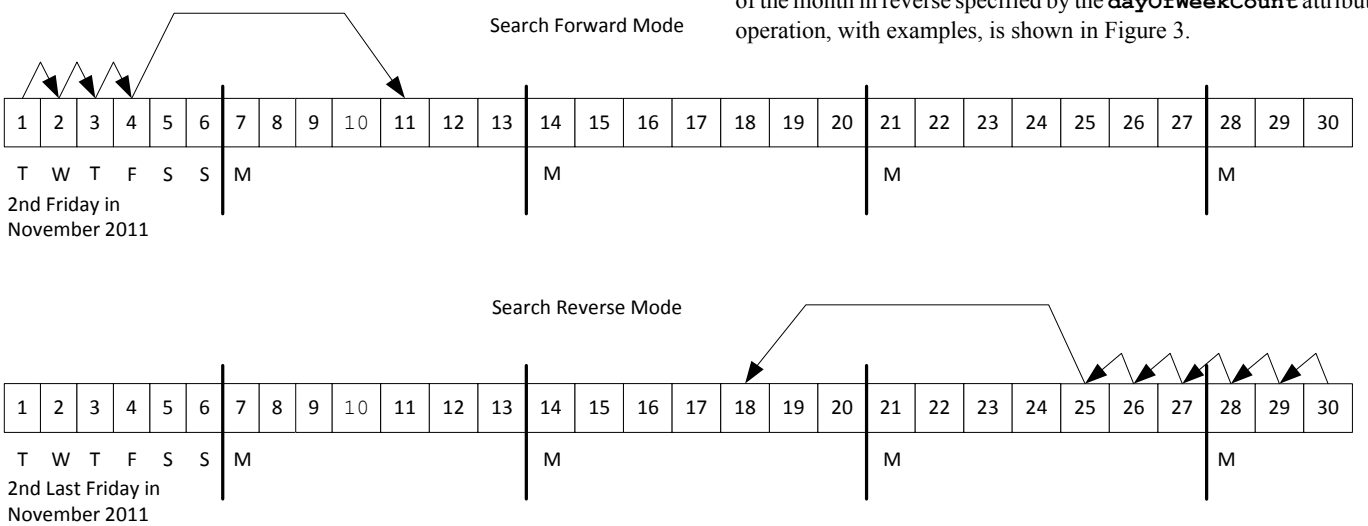
Implementation of **calculateHoliday** in the **DayOfWeekRule** class attempts to provide a date corresponding to the day of the week and the week of the month. Attributes of this class include the **day** of week for which the holiday is to be determined (e.g., Monday = **java.util.Calendar.MONDAY**), the **month** within which this holiday lies and the **dayOfWeekCount** signifying the number of weeks from the beginning (if the **searchForward** flag is true) or the end (if the **searchForward** flag is false) of the month. Thus, 2nd Monday from beginning of May will be represented by an object of this class with the day attribute set to **java.util.Calendar.MONDAY**, **month** attribute set to 5, **dayOfWeekCount** set to 2 and **searchForward** flag set to true. Similarly, the last Thursday of July is represented by an object of this class with the **day** attribute set to **java.util.Calendar.THRUSDAY**, **month** attribute set to 7, **dayOfWeekCount** set to 1 and **searchForward** flag set to false.

Therefore, **calculateHoliday** implementation in **DayOfWeekRule** class has two modes of operation. In the search forward mode, starting from the first day of the specified month it increments date till it finds, for that month, the first day of the week that is specified by the day attribute of the object. Then the increment size increases to 7 and it increments till it finds that day of the week for the week of the month specified by the **dayOfWeekCount** attribute. In the search reverse mode, it starts from the last day of the month and decrements date by 1 till it finds, for the specified month, the last occurrence of the day of the week that matches the value of the **day** attribute of this object. The decrement size is then set to -7. The method then decrements the date till it finds that day of the week for week of the month in reverse specified by the **dayOfWeekCount** attribute. This operation, with examples, is shown in Figure 3.

Listing 4

```
public class DayOfWeekRule
   extends AbstractHolidayRule {
  private boolean searchForward;
  private int dayOfWeekCount;
  private int day;
  private int month;
  public DayOfWeekRule(List<Integer> weekends,
     boolean rollOver, int day, int month,
     boolean searchForward, int dayOfWeekCount){
    super(weekends, rollOver);
    this.day = day;
    this.month = month;
    this.searchForward = searchForward;
    this.dayOfWeekCount = dayOfWeekCount;
  }
  @Override
  protected Date calculateHoliday(int calcYear) {
    int increment = this.searchForward ? 1 : -1;
    Date date = getFirstOccurrence(calcYear,
       increment);
    increment *= 7;
    for (int i = 1;
         i < this.dayOfWeekCount; i++){
      date.add(Calendar.DAY_OF_MONTH, increment);
    }
    return date;
  }
  private Date getFirstOccurrence(int calcYear,
     int increment) {
    Date date= new Date(calcYear, this.month, 1);
    if (!this.searchForward){
      date.add(Calendar.MONTH, 1);
      date.add(Calendar.DAY_OF_MONTH, -1);
    }
    while(date.getDayOfWeek() != this.day){
      date.add(Calendar.DAY_OF_MONTH, increment);
    }
    return date;
  }
}
```

Figure 3



Listing 5

```
public class EasterRule implements HolidayRule {
  @Override
  public boolean isRollOver(){
    return false;
  }
  @Override
  public void interpret(List<Date> holidays,
     int calcYear, List<Date> exclusionDates) {
    Date easterSunday =
       getEasterSunday(calcYear);
    holidays.add(getGoodFriday(easterSunday));
    holidays.add(getEasterMonday(easterSunday));
  }
  private Date getEasterMonday(Date easterSunday)
  {
    Date easterMonday =
       new Date(easterSunday.getYear(),
       easterSunday.getMonth(),
       easterSunday.getDay());
    easterMonday.add(Calendar.DAY_OF_MONTH, 1);
    return easterMonday;
  }
  private Date getGoodFriday(Date easterSunday) {
    Date goodFriday =
       new Date(easterSunday.getYear(),
                easterSunday.getMonth(),
                easterSunday.getDay());
    goodFriday.add(Calendar.DAY_OF_MONTH, -2);
    return goodFriday;
  }
  private Date getEasterSunday(int calcYear) {
    int c, n, k, i, j, l, month, day;
    c = calcYear / 100;
    n = calcYear - 19 * (calcYear / 19);
    k = (c - 17) / 25;
    i = c - c/4 - (c-k)/3 + 19*n + 15;
    i = i - 30 * (i/30);
    i = i - (i/28) * (1-(i/28)
            *(29/(i+1))*((21-n)/11));
    j = calcYear + calcYear/4 + i + 2 - c + c/4;
    j = j - 7 * (j/7);
    l = i - j;
    month = 3 + (l + 40)/44;
    day = l + 28 - 31*(month/4);
    return new Date(calcYear, month, day);
  }
}
```

The final class implementing the **HolidayRule** interface is the **EasterRule** class that is used to determine Good Friday and Easter Monday for a given year (Listing 5).

Good Friday and Easter Monday are not rolled over, therefore, the **isRollOver** method always returns false. **EasterRule**'s implementation of the **interpret** method calls the **getEasterSunday** method to determine Easter Sunday for the given year. **getEasterSunday** is the implementation of the Easter Sunday calculation algorithm of the Astronomical Applications Department of the US Naval Observatory and is taken from their website [1]. Once Easter Sunday is determined, it is incremented to obtain Easter Monday and decremented twice to obtain Good Friday. Both these dates are added to the list of holidays passed to the **interpret** method as an argument.

The **Date** class (Listing 6) used in this application is not the **java.util.Date** class but a simple wrapper over three integers representing day of the month, month of the year (January = 1, February = 2, … December = 12) and the year. It, however, does use **java.util.Calendar** to perform date arithmetic and uses its representation of the days of week. The primary reason for not using **java.util.Date** was to have a very simple representation of date

Listing 6

```
public class Date {
  private int year;
  private int month;
  private int day;
  public Date(int year, int month, int day){
    set(year, month, day);
  }
  public Date(Calendar calendar){
    this(calendar.get(Calendar.YEAR),
         calendar.get(Calendar.MONTH) + 1,
         calendar.get(Calendar.DAY_OF_MONTH));
  }
  private void set(int year, int month, int day){
    this.year = year;
    this.month = month;
    this.day = day;
  }
  public void add(int field, int value){
    Calendar cal = Calendar.getInstance();
    cal.set(this.year, this.month - 1,
            this.day, 0,0,0);
    cal.add(field, value);
    set(cal.get(Calendar.YEAR),
        cal.get(Calendar.MONTH) + 1,
        cal.get(Calendar.DAY_OF_MONTH));
  }
  public int getDayOfWeek(){
    Calendar cal = Calendar.getInstance();
    cal.set(this.year, this.month - 1,
            this.day, 0,0,0);
    return cal.get(Calendar.DAY_OF_WEEK);
  }
  public int getYear(){
    return this.year;
  }
  public int getMonth(){
    return this.month;
  }
  public int getDay(){
    return this.day;
  }
  @Override
  public String toString(){
    return this.day + "-" + this.month
       + "-" + this.year;
  }
  @Override
  public int hashCode(){
    return new Integer(this.year).hashCode() ^
         new Integer(this.month).hashCode() ^
         new Integer(this.day).hashCode();
  }
  @Override
  public boolean equals(Object o){
    boolean reply = false;
    if ((null != o) && (o instanceof Date)){
      Date tmp = (Date) o;
      reply = (this.year == tmp.year);
      reply = reply && (this.month == tmp.month);
      reply = reply && (this.day == tmp.day);
    }
    return reply;
  }
}
```

without time and time zone information but includes the necessary date operations not provided by **java.util.Date**.

The storage and access of holiday rules is beyond the scope of this article. These could be performed in a number of ways ranging from databases,

Listing 7

```
package data;

import java.util.Comparator;

public class DateComparator implements
Comparator<Date> {

  @Override
  public int compare(Date o1, Date o2) {
    String date1 = String.format("%04d%02d%02d",
                  o1.getYear(),
                  o1.getMonth(),
                  o1.getDay());
    String date2 = String.format("%04d%02d%02d",
                  o2.getYear(),
                  o2.getMonth(),
                  o2.getDay());
    return date1.compareTo(date2);
  }
}
```

files or via web services. The access mechanism should return a list of objects each implementing the **`HolidayRule`** interface. Objects that specify rules for holidays that do not roll over need to be at the top of this list for reasons discussed earlier. Additionally, the access mechanism should also return a list of objects of the **`Date`** class representing the exclusion dates. The application should iterate over these rules calling the **`interpret`** method of each rule for a given year. In the end, the application has a list of holidays for the given year that it can use. Output of an example program using the rules in Table 2 (and their implementation discussed above) for 2011, 2012 and 2013 is shown in Table 3.

The dates returned are not in order because the holiday rules that generate them are not processed in the date order but ones for which the holidays do not roll over are interpreted before the others. Ordering can easily be achieved by creating a comparator for the **`Date`** class, assigning an instance of that comparator to a **`java.util.TreeSet`** instance and loading into it the list of holidays returned. **`DateComparator`** (Listing 7) is a comparator to compare two instances of the **`Date`** class. It converts the **`Date`** instances being compared into their string representations in the yyyymmdd format and then returns the result of string comparison. Table 4 shows the dates from Table 3 sorted using a **`java.util.TreeSet`** instance and a **`DateComparator`** object.

Concluding Remarks

Depending on the business requirements, calendar data can be of a considerable size and can have risks associated with errors and omissions in case of manual entry and maintenance. Holiday calendars are a type of calendars that can be dynamically generated from a smaller set of fixed or rarely changing rules, thereby mitigating such risks.

Using the English holiday calendar, this article discusses an implementation of such rules. Implementation of a comprehensive holiday calendar generator is a significant undertaking as these rules and their interpretation can have regional, social and cultural variations. However, a carefully designed hierarchy of representation of these rules can capture most of such variations. Exceptions include rules based on celestial or astronomical events and holidays based on these may have to be explicitly defined. ■

References

[1]The Date of Easter, http://aa.usno.navy.mil/faq/docs/easter.php

[2] Gamma E., Helm R., Johnson R., Vlissides J, (1994), *Design Patterns: Elements of Reusable Object Oriented Architecture*, Addison Wesley

Table 3

2011	2012	2013
29-4-2011	4-6-2012	29-3-2013
22-4-2011	5-6-2012	1-4-2013
25-4-2011	6-4-2012	1-1-2013
3-1-2011	9-4-2012	27-5-2013
30-5-2011	2-1-2012	6-5-2013
2-5-2011	7-5-2012	26-8-2013
29-8-2011	27-8-2012	25-12-2013
26-12-2011	25-12-2012	26-12-2013
27-12-2011	26-12-2012	

Table 4

2011	2012	2013
3-1-2011	2-1-2012	1-1-2013
22-4-2011	6-4-2012	29-3-2013
25-4-2011	9-4-2012	1-4-2013
29-4-2011	7-5-2012	6-5-2013
2-5-2011	4-6-2012	27-5-2013
30-5-2011	5-6-2012	26-8-2013
29-8-2011	27-8-2012	25-12-2013
26-12-2011	25-12-2012	26-12-2013
27-12-2011	26-12-2012	

Code Critique Competition 74

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've tried to write a simply program to split up a single text file into separate files. The idea is each line starting with "--- " and ending with " ---" contains the filename for the lines following it. It doesn't work on my old machine: gives me a 'bad allocation' error. It works on my new machine – although it seems a little slow – but the filenames don't get the trailing minus signs removed. Can you help me find my bug?

Listing 1

```
#include <fstream>
#include <iostream>
#include <string>

int main()
{
  std::ofstream ofs;
  ofs.exceptions(std::ios::failbit);
  try
  {
    std::string lbufr;
    while (std::getline(std::cin, lbufr))
    {
      if (lbufr.find("--- ") == 0 &&
          lbufr.find(" ---") > 0)
      {
        unsigned len(lbufr.find(4, ' ') - 4);
        lbufr.erase(0, 4);
        lbufr.resize(len);
        if (ofs.is_open())
        {
          ofs.close();
        }
        ofs.open(lbufr.c_str());
        continue;
      }
      ofs << lbufr << std::endl;
    }
  }
  catch (std::exception const & ex)
  {
    std::cerr << "Error: " << ex.what();
  }
}
```


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

The example code (unpack.cpp) is in Listing 1 and a sample input file is:

```
--- file1.txt ---
This is file 1
--- file2.txt ---
This is file 2
Line 2
Line 3
```

Critiques

Paul Floyd <Paul_Floyd@mentor.com>

First, let's try to compile it.

```
"test.cpp", line 17: Warning: Conversion of 64
bit type value to "unsigned" causes truncation.
```

That's with Oracle Solaris Studio 12 update 3 beta. GCC 4.5.1 compiles it without a peep.

Give it a quick run. It is 'slow'. Considering what the warning is, I have an idea what is going wrong. Let's fix the warning

```
std::size_t len(lbufr.find(4, ' ') - 4);
```

Now I get an exception.

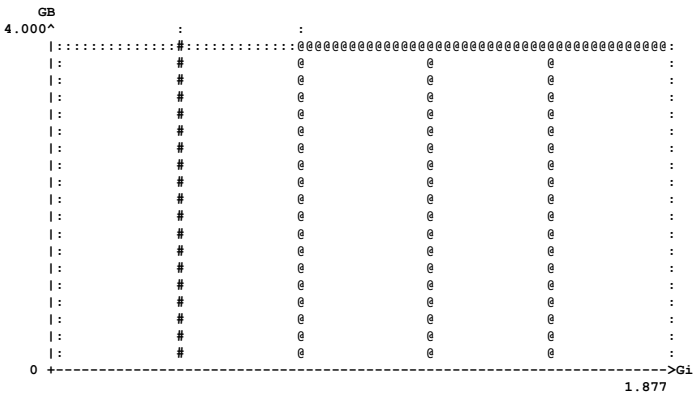
```
Error: invalid string size parameter in function:
basic_string( const _charT*,size_type,const
_Allocator&) size: -5 is greater than maximum
size: -1
```

(without a terminating newline).

Let's go back to the original. I think that the warning is due to the `std::string::find` 'not found' value being truncated to 32bit unsigned, and the string then being resized to that (4Gbytes). Let's confirm:

```
valgrind --tool=massif ./test < in.txt

ms_print massif.out.5706 > ms.out
```

It's a long file, so I'll just show the pretty picture.



X axis is 'giga-instructions'. Peak memory is shown by `#`s, and we see 4Gbytes being allocated.

Revert back to `size_t`, add a couple of debugging `cout` statements, and read the `std::string` doc. The `char` and the `pos` arguments are reversed. The prototype is

```
  size_t find ( char c, size_t pos = 0 ) const;
```

Make that change, and the testcase seems to work.

There are still quite a lot of things that I don't like. The magic numbers 4. Change that.

```
  const char * const prefix = "--- ";
  const char * const suffix = " ---";
  const size_t prefixLength = std::strlen(prefix);
  const size_t suffixLength = std::strlen(suffix);
```

Looking for `" ---"` without checking that it is at the end. I'll use **`rfind`** instead and check that it is at the end.

```
  if ((lbufr.find(prefix) == 0) &&
      (lbufr.rfind(suffix) ==
       lbufr.length() - suffixLength))
```

Using a find for `' '` starting after the prefix won't work if the filename contains spaces. In fact, why do we need to do another **`find`**? We've already done two for the prefix and suffix.

```
  std::size_t len(lbufr.length() - prefixLength
     - suffixLength);
  lbufr.erase(0, prefixLength);
  lbufr.resize(len);
```

The case where the filename is zero length isn't handled correctly (that is, `"---  ---"` in the input stream. Since I've added two calls to **`std::string::length()`**, I can factor that out at the same time.

```
  size_t length(lbufr.length());
  if (length > prefixLength + suffixLength &&
      lbufr.find(prefix) == 0 &&
      (lbufr.rfind(suffix) ==
       length - suffixLength))
```

There are a couple more things that I'd change. I don't like short names like **`lbufr`**. I'd change that to **`lineBuffer`**. The specification of the problem doesn't state what to do in case of an error. In this case, there's a catch all at the top level. Depending on what is really required, it may be better to handle some errors. For instance, if there is a file permission problem, it might be preferable to skip the file and continue. At least some clear message should be printed. As it stands I get **`"Error: iostream object has failbit set"`**. It's also not specified what to do in the case of a zero length filename. Should a warning be printed? In my code below, I just ignore it, and the text is treated as though it were content.

I did think of doing the erase/resize in one go with **`std::string::substr()`**. I doubt that would gain anything, but it is perhaps a little clearer.

From a design point of view, I think I'd almost certainly have chosen XML for the markup rather than `"--- "` and `" ---"`.

So here's my final version.

```
#include <fstream>
#include <iostream>
#include <string>
#include <cstring>

int main()
{
  std::ofstream ofs;
  ofs.exceptions(std::ios::failbit);
  const char * const prefix("--- ");
  const char * const suffix(" ---");
  const size_t prefixLength(
    std::strlen(prefix));
  const size_t suffixLength(
    std::strlen(suffix));
  try
  {
    std::string lbufr;
    while (std::getline(std::cin, lbufr))
    {
      size_t length(lbufr.length());
      if ((length > prefixLength + suffixLength)
           &&
          (lbufr.find(prefix) == 0) &&
          (lbufr.rfind(suffix) ==
           length - suffixLength))
      {
        lbufr.erase(0, prefixLength);
        lbufr.resize(length - prefixLength -
          suffixLength);
        if (ofs.is_open())
        {
          ofs.close();
        }
        ofs.open(lbufr.c_str());
        continue;
      }
      ofs << lbufr << std::endl;
    }
  }
  catch (std::exception const & ex)
  {
    std::cerr << "Error: " << ex.what() << "\n";
  }
}
```

And here is my modified input containing some harder tests: a file called `"---"`, zero length file and empty content.

```
--- file1.txt ---
This is file 1
--- file2.txt ---
This is file 2
Line 2
Line 3
--- --- ---
This is file ---
--- Still file '---'
Last line in file ---.
--- file with blank ---
just one line
---  ---
null filename
--- empty_content ---
--- not_empty ---
not empty
```

Alan Bellingham <AlanB@episys.com>

Let's look at what might cause an issue.

```
>  if (lbufr.find("--- ") == 0 &&
>      lbufr.find(" ---") > 0)
>  {
```

At first sight, this is the line that is most likely to be causing a problem. After all, if this condition fails, then the content of the condition block won't happen.

On examination, the second part of this condition turns out to be unnecessary – it is impossible for it to fail when the first part passes. This is because **`find()`** returns a **`size_t`**, an unsigned value which can therefore only be either equal to or greater than zero. We already know by inspection that it's not zero (or the first part of the condition would have failed), so it must be always true when tested.

So what happens should we encounter a line that starts with `"--- "`, but does not contain the terminating sequence as well, and that in fact has no space later on the line?

Well, in those circumstances, this following line gets interesting. It returns **`size_t(-1)`** to mark failure, but that's a very large unsigned value. Subtracting 4 from it makes it only very slightly less enormous.

```
>  unsigned len(lbufr.find(4, ' ') - 4);
```

```
>  lbufr.erase(0, 4);
```

And now it attempts to change the size of the string. Unfortunately, we're now asking the system to allocate almost all of addressable memory into it, assuming common implementations.

But the complaint isn't about the code misbehaving with malformed data, but misbehaving with data as per the example file. So, given that we're already suspicious about the call to `lbufr.find()` where it's looking for a character, perhaps we ought to check that function.

What does the documentation say?

```
size_type find(charT c, size_type pos=0) const;
```

So, the item to find goes first, and then the optional starting position follows. And the actual call:

```
lbufr.find(4, ' ')
```

Oh dear. Whoops.

This is one of the cases where C++ allows programming errors that other languages would warn about. The first argument is the value 4, also known as CTRL-D in ASCII, and the starting value is ASCII character 32, which is used for the starting position.

So the code ends up looking for CTRL-D (which is almost certainly never within the string it's looking at), starting at position 32, which is quite frequently beyond the end of the string. No wonder the code takes so long: the algorithm is probably optimised on the assumption that it should stop either when the character is found, or when the remaining string size reaches zero. Here, it's starting at less than zero and decrementing each time. It's only going to hit zero when it's wrapped all the way round through the address space and come up to the end of the string from the far side.

Before that happens, there's a pretty good chance that it'll hit some form of access violation and blow up. Modern operating systems are much more likely to enforce this, but older machines without protected memory access were more tolerant.

Of course, when it fails, back comes that `size_t(-1)` value, which blows up the memory manager.

The moral is: be careful to check functions that you call.

Simon Farnsworth <simon@farnz.org.uk>

The big clue from the student is 'it doesn't work...gives me a "bad allocation" error'. Something in this code allocates memory far in excess of what you would expect.

The code compiles without error or warning (from `gcc -Wall`), so the bug is going to be subtle. So, first target is the allocation of memory that's implicit in `lbufr.resize` – if `len` is completely crazy, that could lead to the bug. We could simply print `len`, and see if that exposes the bug, but I'm going to take us down a different route that leads to the compiler finding the bug for us.

As a first step, let's stop manipulating the same buffer – we'll treat it as a constant throughout the loop. Some discipline is required here, but we'll get the compiler to help. As a side-effect of doing this, we will need to pull the filename out into its own variable. We also spot that `len` is never deliberately changed, so should be `const` to get the compiler to inform us if we're wrong. Change the body of the loop to:

```
const std::string &line(lbufr);
if (line.find("--- ") == 0 &&
    line.find(" ---") > 0)
{
  const unsigned len(line.find(4, ' ') - 4);
  std::string filename(line);
  filename.erase(0, 4);
  filename.resize(len);
  if (ofs.is_open())
  {
    ofs.close();
  }
  ofs.open(filename.c_str());
  continue;
}
ofs << line << std::endl;
```

We no longer refer to `lbufr` inside the loop, instead referring to our new const reference `line`. There's no change to behaviour, yet, but there's now an obvious silly thing – why are we copying the string, only to delete its beginning and end? If we can make `filename` a copy of a substring of the line, we are able to make it `const`, too. While we're at it, `len` is a size, and would more normally be a `size_t` in library functions. This gets us:

```
const std::string &line(lbufr);
if (line.find("--- ") == 0 &&
    line.find(" ---") > 0)
{
  const size_t len(line.find(4, ' ') - 4);
  const std::string filename(
    line.substr(4, len);
  if (ofs.is_open())
  {
    ofs.close();
  }
  ofs.open(filename.c_str());
  continue;
}
ofs << line << std::endl;
```

Now we see a behaviour change; the program runs much faster, but it still puts the text in files without the trailing `" ---"` removed. Looking at the documentation for `substr`, this is what happens if we ask for a substring bigger than the entire source string, telling us that `len` is definitely too large.

Let's continue down the path of moving things to clearly named `const` variables; we introduce constants for the filename prefix and suffix, and use them instead of the literals. It's obvious that the literal 4s are meant to be the lengths of the prefix and suffix, and that `len` is meant to be the length of the filename. Less obvious (but still true) is that the literal space is meant to be the filename suffix. We rename `len` to `filename_length`, for clarity, and use our new constants for the prefix and suffix:

```
const std::string &line(lbufr);
const std::string filename_begin("--- ");
const std::string filename_end(" ---");

if (line.find(filename_begin) == 0 &&
    line.find(filename_end) > 0)
{
  const size_t filename_length(line.find(
    filename_begin.length(), filename_end) -
    filename_begin.length());
  const std::string filename(line.substr(
    filename_begin.length(), filename_length));
  if (ofs.is_open())
  {
    ofs.close();
  }
  ofs.open(filename.c_str());
  continue;
}
ofs << line << std::endl;
```

This stops compiling – we've found the bug. The arguments to `find()` are swapped; this explains the symptoms, as `find()` is specified to return `std::numeric_limits<size_t>::max()` if the string is not found, resulting in length being insanely large.

It also explains why the behaviour is different on different systems; on my 64-bit system, unsigned is 32 bits large, while `size_t` is 64 bits large. Integer coercion drops the top 32 bits of the result of `find()`, resulting in my machine trying to allocate a shade under 4GB for each filename, which takes a while. On a 32-bit system, where `size_t` is 32 bits, it would

still try and allocate just under 4GB, but this time it would fail with a bad allocation error.

We swap them to the correct order, and the program now works for the sample input. There's still a lurking bug, though; we have assumed that no filename contains the substring `" ---"`, despite this not being in the student's spec. This is trivial to fix – changing **`find`** to **`rfind`** results in the last match being found, and we then just need to compare this to the length of the input line to be confident we've found a match. At the same time, we no longer need two arguments to **`rfind`**, so let's just remove the start position argument completely:

```
const std::string &line(lbufr);
const std::string filename_begin("--- ");
const std::string filename_end(" ---");

if (line.find(filename_begin) == 0 &&
    line.rfind(filename_end) ==
      (line.length() - filename_end.length()))
{
  const size_t filename_length(line.rfind(
    filename_end ) - filename_begin.length());
  const std::string filename(line.substr(
    filename_begin.length(), filename_length));
  if (ofs.is_open())
  {
    ofs.close();
  }
  ofs.open(filename.c_str());
  continue;
}
ofs << line << std::endl;
```

We now have code that meets the specification given, but it's ugly, and therefore going to be tricky to maintain. There are two problems, in my view; firstly, there's a **`continue`** buried deep in the **`if`** statement. As a rule of thumb, **`continue`** and **`break`** from loops should only be used when checking preconditions at the beginning of a loop; they cause surprises if they come after the loop has done significant amounts of processing. In this case, it's easy to remove it – just change it for an **`else`**.

Secondly, we have two large lumps of code whose purpose is not completely obvious, and in which we've already found bugs; one is the code to determine if this is a filename, the other is the code to extract the filename from a line. These should be extracted into clearly named functions; it's generally a good rule to optimize source code for readability first, performance second, only breaking this rule where a profiler or similar measuring tool tells you that the clear code is unacceptably slow.

My final code becomes:

```
#include <fstream>
#include <iostream>
#include <string>
#include <cassert>

namespace
{
const std::string filename_begin("--- ");
const std::string filename_end(" ---");

const bool isFilename(const std::string &line)
{
  return line.find(filename_begin) == 0 &&
         line.rfind(filename_end) ==
        (line.length() - filename_end.length());
}

const std::string extractFilename(const
  std::string &line)
{
  assert(isFilename(line));

  const size_t filename_length(line.rfind(
    filename_end ) - filename_begin.length());
  return line.substr(filename_begin.length(),
    filename_length);
}
}

int main()
{
  std::ofstream ofs;
  ofs.exceptions(std::ios::failbit);
  try
  {
    std::string lbufr;
    while (std::getline(std::cin, lbufr))
    {
      if (isFilename(lbufr))
      {
        if (ofs.is_open())
        {
          ofs.close();
        }
        ofs.open(
          extractFilename(lbufr).c_str());
      }
      else
      {
        ofs << lbufr << std::endl;
      }
    }
  }
  catch (std::exception const & ex)
  {
    std::cerr << "Error: " << ex.what() <<
    std::endl;
  }
}
```

You will notice that I've made a few small changes not discussed above. The assertion in **`extractFilename`** is simply because that's previously been buggy code; if nothing else, the assert draws your attention to a precondition that's not otherwise documented. I've put **`std::endl`** on the exception handler, to make the output easier to read.

Finally, I've put my utility functions and constants into an anonymous namespace; this is intended to help any future maintainer, by indicating that I only intended these constants and functions to be used within this file. As a general rule, it's worth getting into the habit of using namespaces whenever appropriate; they stop symbol collisions, and provide useful information to the maintainer.

Matthew Wilson <stlsoft@gmail.com>

Superficial Inspection

Before investigating the semantics in detail, I gave the code a superficial inspection and made the following observations on the code as presented:

- turning on **`IOStreams`**' exceptions is a place where dragons lie – something to check on later?
- why call the variable **`lbufr`** and not simply (and more clearly) **`line`**?
- the return type of **`basic_string<>::find()`** is an unsigned integer, and the 'not-found' sentinel **`basic_string<>::npos`** is -1 of that type, which is equivalent to **`std::limits<size_type>::max()`**, so it's always going to be non-negative. Given the second string cannot be found at index 0 if the first is, the second conditional subexpression **`lbufr.find(" ---") > 0`** is effectively always true, so the conditional expression depends only on whether the line begins with `"--- "`. This opens the possibility of a horrible fault – of which more later;
- use of constructor syntax for initialisation of **`len`** seems a bit obtuse;

- the nature of the evaluation of **len** precludes use of paths with spaces, and will fail in such cases. Furthermore, there are possibilities of two faults. The first is similar to the earlier one, with the same awful unintended consequences. The second could be the explanation of your memory issue – of which more later;
- the manipulation of **lbufr** is baroque. Why not declare and initialise a new (immutable) **std::string** instance via a single call to **substr()**?
- why **continue**, rather than just **else**?
- why use **std::endl**? Is it necessary to ensure that each line of each output file is written wholly in the case where the total file is not written wholly? If not necessary, then it is a needless performance cost, since it effectively circumvents the buffering of the Standard C++ IOStreams library (and the presumable underlying Standard C Streams) file layer;
- catching **std::exception** is an appropriate last-resort exception-handler, but I suspect this will be inadequate to issue an appropriate contingent report in the case of file-system failures;
- the program does not have any explicit return value, so will always (implicitly) return 0, indicating success in all cases. Given that a catch-clause exists, and its form implies non-normative behaviour, **EXIT_FAILURE** should be returned here.

Furthermore, in terms of functionality I make the following further observations:

- I think the program should, by default, refuse to expand/decompress into a file that already exists. A program option should be available to effect overwrites only under explicit user direction;
- I think the program should be able to take its input either from the standard input stream or from a named file;
- the (regex) format of /^--- (.+) ---$/ of the sentinel line is only moderately unique. False positives (even without the defects) are not very unlikely. Of course, if this is the publicly documented behaviour of the program, then no user has cause to complain. But they may have cause to not use it, if its bad design causes it to be unusable.

Let's deal with the defects first. In reverse order (because they are dependent):

3. memory exhaustion;

2. match to non-sentinel lines;

1. obtain wrong output file path (and potentially overwrite unintended file).

Smoke test(s)

Next, I ran some smoke tests. I created the file `file1.txt`, whose contents are as follows (with \t indicating a TAB character and \n indicating a line ending):

```
--- file1-a.txt ---\n
abc\t\n
--- abc\n
def        \n
\n
--- file1 b.txt ---\n
ghi\n
jklm  \n
\n
nop
```

When I run with the original program with this input the process throws an exception in the call to **resize()**. This is a consequence of a defect I hadn't spotted during visual inspection: the arguments presented to the **find()** method are in the wrong order: the first parameter is the character; the second the start position. What's actually requested is the index of the character **'\4'**, which doesn't occur anywhere in the input file (and is not likely to occur, as it's the End-Of-Transmission character in ASCII), from position 32 (the value of the space-character code). In pretty much every line it'll encounter, the **find()** will fail, and will return **npos** (e.g. 0xffffffff, 0xffffffffffffffff), from which 4 is subtracted to evaluate **len**. Consequently, the call to **resize()** fails.

Fix-1

Obviously, this is defect must be fixed, yielding the second version of the program (program0.fix1), as in:

```
/* file: program0.fix1 */
...
  unsigned len(lbufr.find(' ', 4) - 4);
...
```

Running this program immediately yields another failure. When the third line, "**--- abc**", is encountered, the problem with the second sub-expression manifests. The non-sentinel **sline** is matched as a sentinel line, but since it does not contain a space after index 4, the **find()** fails and **len** is again a very large number, and **resize()** fails.

Fix-2

The second fix is to correct the second sub-expression, yielding the third version of the program (program0.fix2), as in:

```
/* file: program0.fix2 */
...
  lbufr.find(" ---") == lbufr.size() - 4)
...
```

Running the program passes the previous defect, but fails on the second proper sentinel line, the one with the space in the path. It produces a file with the name "file1", rather than the required "file1 b.txt".

Fix-3

The fix is to simplify the evaluation of **len**, dispensing with the unnecessary search for something we've already (correctly, as of the second fix) evaluated, namely the start of the trailing sentinel sequence. This yields the third version of the program (program0.fix3), as in:

```
/* file: program0.fix3 */
...
  unsigned const len = lbufr.size() - 8;
...
```

This version works for the full input test case file file1.txt. Of course, that's hardly exhaustive, but at least we've exercised all the defects I spotted (and the one I didn't!); in a 'proper' development, I'd have extracted the parsing code to a function and subjected it to extensive unit-tests.

However, I think we can simplify the evaluation of the file name, and dispense with **len** altogether, using **substr()**. Along with the other minor nits from my first inspection, I would instead rework the program (program0.fix4) as follows, with the changed parts darker:

```
/* file: program0.fix4 */
#include <fstream>
#include <iostream>
#include <string>
#include <cstdlib>

int main()
{
 std::ofstream ofs;
 ofs.exceptions(std::ios::failbit);
 try
 {
   std::string line;
   while (std::getline(std::cin, line))
   {
     if (line.find("--- ") == 0 &&
         line.find(" ---") == line.size() - 4)
     {
       std::string const path =
         line.substr(4, line.size() - 8);
       if (ofs.is_open())
       {
         ofs.close();
       }
```

```
        ofs.open(path.c_str());
      }
    else
      {
        ofs << line << "\n";
      }
    }
  }
  catch (std::exception const & ex)
  {
    std::cerr << "Error: " << ex.what();

    return EXIT_FAILURE;
  }

  return EXIT_SUCCESS;
}
```

Inadequate failure-handling

Consider now a second input file, file2.txt, which has a slight addition, the line `"---  ---"`:

```
--- file1-a.txt ---\n
abc\t\n
--- abc\n
def        \n
\n
--- file1 b.txt ---\n
ghi\n
jklm  \n
---  ---\n
\n
nop
```

When run with program0.fix4, we find that the new line precipitates an exception, thrown during the call to **`ofs.open()`**. However, one of the lacklustre areas of the standard library – failure handling – is brought to light here: with Visual C++ (various versions), the catch-clause causes the almost-completely-useless contingent report **`"Error: ios::failbit set"`**; with GCC (3.4, on Windows) it gives the even less useful **`"Error: basic_ios::clear"`**.

This issue is part of a much larger set of issues that I'll be dealing with in the next few instalments of my reinvigorated 'Quality Matters' column, as of next month's *Overload*, so I won't go into it any further here beyond saying that a pre-emptive check should be made on the path; in this particular case it would be easy to specify a useful contingent report as in:

```
...
  if (ofs.is_open())
  {
    ofs.close();
  }
  if(path.empty())
  {
    std::cerr << "Error: missing file"
      "name/path\n";
  }
  ofs.open(path.c_str());
...
```

In fact, given the scope of the program, I would prefer to implement it in C, because that makes the failure-handling much more clear and straightforward! I'll discuss that further in the next QM, wherein I'll present several alternatives, including the C version.

(Pre-emptive thanks to Roger for being kind enough to agree to my re-using this example program later.)

Barry Nichols <barrydavidnichols@gmail.com>

The main problem with this code is at the line:

```
unsigned len(lbufr.find(4, ' ') - 4);
```

The first parameter of the **`string::find()`** function is the string to find and the optional second parameter is the position to start searching at. However, in the example code the parameters are the wrong way round. But as the example code used a **`char ' '`** it was converted to an **`int`** and used as the starting location.

The corrected code would be:

```
unsigned len(lbufr.find(' ', 4) - 4);
```

But this is only part of the problem as the there may be a space in the file name at which the above will return the position of the space rather than the end of the file name, this can be remedied by using the string we want to match as the first parameter to **`find()`** which is **`" ---"`**:

```
unsigned len(lbufr.find(" ---", 4) - 4);
```

Another problem is that **`lbufr.find(" ---") > 0`** will always be true as find returns an unsigned value, this will result in the program interpreting any line containing **`"---  "`** as a file name, regardless of whether or not it contains **`" ---"`**. The correct way to check if the string wasn't found is to compare the resulting position to **`std::string::npos`**, which is the value returned by **`find`** when the search string is not found:

```
lbufr.find(" ---") != std::string::npos
```

Also much of this code is unnecessary as the **`substr()`** member of the string class can be used to get the filename e.g.:

```
lbufr.substr(4, lbufr.find(" ---", 4) - 4);
```

By using the **`c_str()`** member to convert the returned value to a cstring this can be used directly as an argument to **`ofs.open()`**:

```
ofs.open(lbufr.substr(4, lbufr.find(" ---",
   4) - 4).c_str());
```

Which would remove a few lines of code. The **`continue`** statement can also be removed if the line **`ofs << lbufr << std::endl;`** is placed inside an **`else`** which will make the code clearer as it will be obvious that the code is either opening a new file IF a filename is given ELSE writing to the open file.

Also, the files are only closed when a new file name is found so the last file doesn't get closed. It would be better to close the last file at the end of the program, by simply repeating:

```
if (ofs.is_open())
  ofs.close();
```

after the **`catch`** block.

The resulting code after these alterations would therefore be:

```
#include <fstream>
#include <iostream>
#include <string>

int main()
{
  std::ofstream ofs;
  ofs.exceptions(std::ios::failbit);
  try
  {
    std::string lbufr;
    while (std::getline(std::cin, lbufr))
    {
    if (lbufr.find("--- ") == 0 &&
        lbufr.find(" ---") != std::string::npos)
    {
      if (ofs.is_open())
      {
        ofs.close();
      }
      ofs.open(lbufr.substr(4,
        lbufr.find(" ---", 4) - 4).c_str());
    }
    else
    {
```

```
      ofs << lbufr << std::endl;
    }
  }
  }
  catch (std::exception const & ex)
  {
    std::cerr << "Error: " << ex.what()
      << std::endl;
  }
    if (ofs.is_open())
  {
    ofs.close();
  }

  return 0;
}
```

Graham Patterson <grahamp@berkeley.edu>

If I was faced with this sort of problem in my normal work, I would consider the Unix utilities **split** or **csplit**. However both of these are intended to split the input into predefined named files without extracting any lines. The problem, as gleaned from the example code, is to remove the filename marker lines and separate the remaining text into the provided file names.

I will also preface the following comments with the observation that I do not work in C++. The general algorithm seems reasonable:

1. Read the input from **stdin**
2. If the line is a file name marker, extract the name. If there is an existing output file stream active, close it before opening the new file for output.
3. Write non-marker lines to the current output stream.

There are a couple of issues with the design as implemented. First, the final output file is not closed. This would be a problem if the code was used as a frequently called routine, as most environments have an upper bound on active file streams. Secondly, the code assumes that the first line of input is a valid file marker line. This is a little risky in the general case, though if the input is machine generated it might be considered reliable. Without knowing the design parameters it is difficult to assess the risk in this instance.

On to the code provided. The marker line test is weak. It is possible that it would match a line without a valid file name. `'--- ---'` meets the criteria, for example. The second clause should be

```
lbufr.rfind(" ---") > 6
```

to ensure that the tail of the marker allows space for a filename of at least one character. Since the marker strings are both four characters, the code would be more robust with the strings assigned to variables, and the magic 4s converted to the length of these strings.

The removal of the marker text needs to be re-thought. The test for the marker start and end is more reliable than just stepping over the presumed filename because it is anchored at the start and end of the line. They both have to be present for a valid marker line. The leading and trailing markers can be removed with variants of the **.erase()** method. Since there are no specific notes on the file name format, the possibility of spaces in the file name should be considered when using the **.find()** method, and the residual filename should be at least one non-space character. A robust approach would rule out pure path separators as well, but this goes back to the parameters governing the generation of the source file. Tightening up the file name extraction should remove the run-time errors which are probably due to this string processing.

This is one of those tasks that I would probably give to AWK or Python to handle. These languages have the benefit of simple regular expression pattern matching and easy removal of unneeded fields.

Here is an AWK solution:

```
function closefile(f) {
  if(f != "") {
    close(f)
  }
}
  BEGIN  {
    outfile =""
  }

  $1 ~ /---/ && $NF ~ /---/ && NF >= 3 {
    gsub(/^--- +/, "")
    gsub(/ +---$/, "")
    closefile(outfile)
    outfile = $0
    next
  }

  outfile != "" {
    print $0 > outfile
  }

  END  {
    closefile(outfile)
  }
  #
```

and here is a Python one:

```
import sys
import re

pattern = re.compile(r"^--- (.+) ---$")
f = False
for l in sys.stdin:
  fn = re.match(pattern, l)
  if fn:
    if f and not f.closed:
      f.close()
    f = open( fn.group(1),'w')
    continue

  f.write(l)
else:
  if f and not f.closed:
    f.close()
```

Huw Lewis <huw.lewis2409@gmail.com>

My laptop reports the **bad_alloc** exception when running the program as is. This suggests either a huge amount of memory is being requested or some kind of infinite loop that exhausts the memory available to the process. The error occurs very quickly so I expect it is the former.

The main function has no return statement. Most compilers let us get away with that and provide a zero return code for us. But in our case wouldn't we want to return a non-zero (error) code on failing to execute successfully? Our exception handler represents the failure case so why not set an error code here?

```
catch (std::exception const& ex)
{
  std::cerr << "Error: " << ex.what();
  return -1;
}

return 0;
```

The **getline** function is declared in **<string>** as a helper method and extracts data from an **istream** up to (but not including) an end of line character. The **getline** function returns a reference to the **istream** object and this is being used as the **while** loop condition. The stream (**ios**) base class provides an overloaded **operator void* ()** which returns a **NULL** pointer if any of the error status flags are set. Once we get

to the end of the file and do another getline operation this will become the case and loop will terminate.

The two **string::find** operations tell us that the line begins with "--- " and also has another " ---" marker later on in the line and therefore this is a file boundary marker line. Therefore we should close any existing output file data and start a new one.

A third **string::find** operation is performed on the line buffer in order to calculate the length of the filename portion. The bug is here – the parameters are the wrong way around. The author meant to tell the operator to start the search for a space character (value = 32) from index 4. Instead we have a search for the value 4 from the 32nd character – which clearly doesn't exist and will instead return the **std::string::npos** constant to indicate failure. This constant is of type **size_t** (unsigned) but is defined as **-1** to result in the maximum value for **size_t** i.e. very, very large.

Crucially the code does not check the success of this **find** operation and uses the result in deriving the **len** value. The **lbufr** string is modified to erase the characters leading up to the file name, then resized to the file name length. However, the error in calculating the file name length results in a request to allocate a gigantic string and this is the source of the **bad_alloc** exception.

One solution is to all this is to use the result of the 2nd find to eliminate the need for the 3rd. This result could be used to create a new string representing the file name without the need for the **erase** and **resize** operations. I've re-written this into a helper function as follows:

```
bool isFileBoundaryLine(const string& line,
  string& fileName)
{
  bool isBoundaryLine(false);
  if (line.find("--- ") == 0)
  {
    // check for the end marker
    const size_t endIndex = line.rfind(" ---");
    if (endIndex != string::npos)
    {
      isBoundaryLine = true;
      const size_t startIndex(4);
      fileName.assign(&line[startIndex],
          endIndex - startIndex);
    }
  }
  return isBoundaryLine;
}
```

There are still a couple of things I'm not so keen on. The fixed format of the file name line is a little fragile. Maybe the use of the formatting stream operations could help handle erroneous whitespace.

The continue statement is only one step away from **goto**. It isn't a good way to structure logic and can be eliminated with a simple **else** statement. After a little more tidying, my final version is:

```
#include <fstream>
#include <iostream>
#include <string>
#include <sstream>

using namespace std;

bool isFileBoundaryLine(const string& line,
   string& fileName)
{
  bool isBoundaryLine(false);
  if (line.find("--- ") == 0)
  {
    // check for the end marker
    const size_t endIndex = line.rfind(" ---");
    if (endIndex != string::npos)
    {
      isBoundaryLine = true;
      const size_t startIndex(4);
      fileName.assign(&line[startIndex],
          endIndex - startIndex);
    }
  }
  return isBoundaryLine;
}

int main() {

  ofstream ofs;
  ofs.exceptions(std::ios::failbit);
  try
  {
    string lbufr;
    while (getline(std::cin, lbufr))
    {
      string fileName;
      if (isFileBoundaryLine(lbufr, fileName))
      {
        if (ofs.is_open())
          ofs.close();
        ofs.open(fileName.c_str());
      }
      else
      {
        ofs << lbufr << std::endl;
      }
    }
  }
  catch (exception const& ex)
  {
    cerr << "Error: " << ex.what();
    return -1;
  }

  return 0;
}
```

Finally, I believe in choosing the best tool for the job at hand, and in this case I'm inclined to suggest Python as a better candidate for this job. The same program can be expressed in just a few easy (and customisable) lines as shown below:

```
#!/usr/bin/python
import sys

if __name__ == '__main__':
  f = None
  for line in sys.stdin:
    if line.startswith('--- ') and line.
      endswith(' ---\n'):
      filename = line.strip('- \n')
      f = open(filename, 'w')
    elif f is not None:
      f.write(line)
```

Commentary

The original code demonstrates some of the problems that can be caused when the size of integer data types are different on different platforms. Many people are very used to coding on a single platform and may become unaware of differences between the various sorts of integers. Unfortunately they are not interchangeable as this example demonstrates.

The second problem is the failure to detect the arguments being passed in the wrong order. This is surprisingly hard to spot because of the silent conversion between integer and char data types. However, the same problem occurs in other languages whenever both arguments are of the same type and you just get the order wrong.

Time For A Change?

Ric Parkin feels some 'new blood' is needed to help keep Overload at the top.

Over four years ago I was asked, along with a few others, to guest edit an issue of *Overload* while Alan Griffiths took a well-earned break. I did *Overload* 81 [1].

Helped by the team, it was a fun time and a great experience, and a few months later I took over as the permanent editor. I've enjoyed it immensely since and have got a lot out of it, but think it's time to start looking for a new editor.

So what's involved? Most important is coordinating between the various parties. It all starts very early in the process, as you're the point of contact for authors – they submit an article or just suggest an idea. After giving some early feedback and getting a good solid revision, you then send the article to the review team and collate their suggestions (as well as reviewing it yourself) ready to return to the author. Occasionally an article needs some more hands-on-changes – perhaps a non-native speaker needs language help – and you get someone to work closely with the author. Once the articles are ready, they go to the production editor, who sets them in our house style and returns the proofs. At this stage minor corrections can be done, subtitles and pullquotes chosen, and tweaks to layout performed to make it look good. Your biggest job happens around this time – writing the editorial. I can sometimes find this a little daunting if I haven't thought of a good subject, but in reality you can easily adjust the content as you see fit. Once it's gone to the publishers you can have a rest, until it's time to start looking at the articles for the next issue.

This whole cycle is spread over two months so it's not an onerous amount of time – if you're organised and prompt it only usually takes a few minutes here and there. And you really do get something special out of it – it looks great on your CV, and you get to talk to some really great authors.

Interested, or just want more details? Just ask me or any other committee member – a chat over a beer at the conference would be great opportunity. Being a guest editor for an issue would also be a great introduction to what's involved and can be easily arranged – why not find out?

Reference

[1] http://accu.org/index.php/journals/c232/

RIC PARKIN

Ric has been programming professionally for around 20 years, mostly in C++, for a range of companies from tiny startups to international corporations. Since joining ACCU in 2000, he's left a trail of new members behind him. He can be contacted at ric.parkin@gmail.com.

Code Critique Competition (continued)

Checking the return code from the function would have detected the problem earlier. I won't add any more as the entrants covered a lot of ground between them!

The Winner of CC 73

It was a hard choice this time – partly because of having a good number of entries. I decided, after some musing, that Paul Floyd's answer was the best one this time because of the combination of use of a tool, the commentary and his test cases.

Code Critique 74

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

A classic little problem here: can you explain what might be problematic about the class in Listing 2, recently found in an actual production code base…

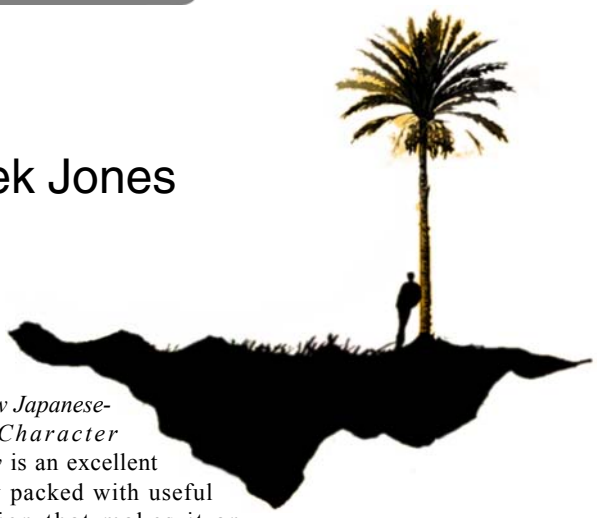
You can also get the current problem from the accu-general mail list (next entry is posted around the last issue's deadline) or from the ACCU website (http://www.accu.org/journals/). This particularly helps overseas members who typically get the magazine much later than members in the UK and Europe.

Listing 2

```
/**
  Singleton template definition
*/
#ifndef _SINGLETON_H
#define _SINGLETON_H

namespace utilities
{
  /** Singleton template */
  template<class T> class Singleton
  {
  public:
    static T* getInstance()
    { return theInstance; }
    static void setInstance(T *instance)
    { theInstance = instance; }

  protected:
    Singleton();
    ~Singleton();

    static T* theInstance;
  };
} // namespace utility

#endif // _SINGLETON_H
```

Desert Island Books

Francis Glassborow maroons Derek Jones on our desert island.



I have been asked to write a few words introducing Derek. First let me warn you to be careful if you google for him. Try 'Derek Jones C programming' and you will discover two people meet these criteria and both are on the academic side. Our Derek is the one whose company is Knosoft. He is also the author of arguably the most comprehensive book on C that has ever been written. Titled *The New C Standard: An Economic and Cultural Commentary*, this tour de force was eventually rejected by his publisher and is now available for free as an ebook (pdf format).

I can remember him taking time to explain to me why programming language standards are important in ways that are unlike standards for electric wiring (mainly concerned with safety issues).

Derek is very different from your average programmer in that he extends his interests into many aspects of code writing that are usually ignored by others. He certainly has a strong academic bent and has a high reputation in the world of programming Standards. He is also concerned with what causes errors in code. He has spent time and effort trying to determine the causes of coding errors.

Some of you may have met him at one of our conferences where he has taken the opportunity to do various pieces of research into the psychology of programming and programmers. If you do not already follow his blog, I think you will find it worth surfing to www.knosof.co.uk and then follow the link to 'The Shape of Code'.

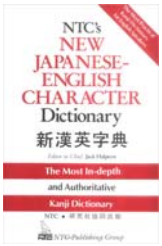
Finally, I think that Derek is one of those people who might enjoy the peace of a desert island sitting in the sun away from distraction and thinking about the many things that influence the quality of code.

Derek M Jones

Why would I want to take books to a desert island? Two reasons that spring to mind are because I get pleasure reading them and because they contain information that is not in my head.

I am a plot oriented reader and having enjoyed reading a book have to wait a good number of years before there is a chance I will enjoy it as much on a second reading. Taking books for pleasure that I have not yet read is a risky business, there is a high probability I will not enjoy them and besides reading them once every few years is an inefficient use of resources.

The one of the few kinds of book that both pleasureable to leaf through and contain lots of information are dictionaries. With the economic and cultural center of gravity moving to China it would be useful to be able to read Chinese. NTC's *New Japanese-English Character Dictionary* is an excellent dictionary packed with useful information that makes it an interesting read. This dictionary choice is based on ignorance about good Chinese dictionaries (I have not looked for any) and because I know that Japanese Kanji characters are mostly taken directly from Chinese and Japanese people I know tell me they can read a lot of Chinese text.

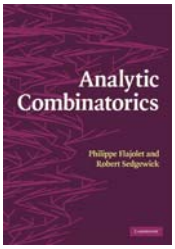


I could spend time on the beach drawing and learning a couple of characters a day.

So what other information would I like to acquire and what are the good books on those subjects?

If my island includes a computer (and the necessary solar or hydroelectric power) I could write some software. I stopped being at the sharp end of compiler writing at about the time that commonly available processor hardware started to do lots of very weird and wonderful stuff internally (e.g., chopping instructions up into smaller micro operations that executed in who-knows-what order and shadow register units). It would be fun to write an optimizing code generator that took these features into account. So I would need a copy of the processor reference manual for the cpu in the computer I had (for once I would be happy for this to be Intel because their processors invariable have strange quirks and I have time on my hands rather than having to ship to customers yesterday).

If I was stuck with a computer and no processor reference manual (or the wrong one) I could always work on a compiler front end. Minimalism and C++ are two topics that rarely appear together and if I expected to spend many years on my island a minimalist C++ compiler front end would soak up the time. Ok, yes, the C++ Standard is a hastely written bloated sprawl of a language, but compiler writers are hired guns who would rarely work if they limited themselves to languages they thought worthy of their time. I need a challenge and writing a minimalist C++ front end is certainly that, Java would not take that many years, the world needs me to invent a new language like I need a hole in the head and just think of the press coverage 'Man rescued from desert island wrote C++ front end in 100,000 lines of code'.



There will be lots of grains of sand to count on my desert island and one book that I would like to have the time to complete is *Analytic Combinatorics* by Philippe Flajolet and Robert Sedgewick. Combinatorics is the science of combinations and finding a structure to the patterns and sequences of trees, leaves, sand and other things I see everyday would allow me to delude myself into thinking that I understand the world I live in. This book is at the upper end of my mathematical abilities and so will also help me stay on my toes mentally.

Building items for my island will require tieing things together in such a way that they don't slip and slide. The Japanese have developed a sophisticated art of rope binding (Kinbaku, in the west Shibari is the commonly used term). However, books associated with this art are more likely to target readers interested in Japanese bondage and BDSM than

What's it all about?

Desert Island Disks is one of Radio 4's most popular and enduring programmes. The format is simple: each week a guest is invited to choose the eight records they would take with them to a desert island (http://www.bbc.co.uk/radio4/factual/desertislanddiscs.shtml).

The format of 'Desert Island Books' is *slightly* different from the Radio 4 show. You choose about five books, one of which must be a novel, and up to two albums. Some people even throw in the odd film. Quite a few ACCUers have chosen their Desert Island Books to date and there are plenty more to go.

The rules aren't too strict but the programming books must have made a big impact on your programming life or be ones that you would take to a desert island. The inclusion of a novel and a couple of albums helps us to learn a little more about you. The ACCU has some amazing personalities and Desert Island Books has proved we only scratch the surface most of the time.

Each issue of CVu will have someone different. If you would like to share your Desert Island Books please email me: paul.grenyer@gmail.com.

Bookcase

The latest roundup of book reviews.

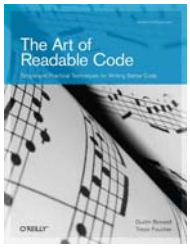
If you want to review a book, your first port of call should be the members section of the ACCU website, which contains a list of all of the books currently available. If there is something that you want to review, but can't find on there, just ask. It is possible that we can get hold of it.

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

I must thank Blackwells and Computer Bookshop for their continued support in providing us with books.

Jez Higgins (jez@jezuk.co.uk)

The Art of Readable Code



By Custin Boswell and Trevor Foucher, published by O'Reilly, ISBN 0596802293

Reviewed by Paul Floyd

This is a short book. Amazon count 204 pages, but I reckon that if you only count pages with stuff to read on, there aren't much over 100 pages. So I was a little amused when the authors stated that they expected you to read the book in a week or two. 3 days at a push more like it.

So, it's clearly not a subject that is amenable to inspire a 1000 page tome. There are quite a few cartoons which, depending on your point of view, pad out the pages or add some levity.

I felt that the authors were struggling to stretch the idea to fill a book. The advice on 'plain English' for comments and variable names is sound. I wasn't sure sometimes whether the ideas about refactoring were 'readability' issues or just simply better code.

The example code leans towards Google Javascript. That made me think a bit about Spinellis' *Code Reading* which, though coming at the problem from the other direction (how to read code rather than how to write code that can be read), does have fairly broad coverage. I don't think that you can have a checklist for the creativity required for readability, but perhaps this book would have benefited from a few pearls of wisdom from a few revered programming gurus.

The Boost C++ Libraries

By Boris Schäling, published by XML Press, ISBN 0982219199

Reviewed by Paul Floyd

On the whole I enjoyed reading this, and found the coverage quite broad. Several of the Boost libraries that are described I've never used and so I was interested to see what they can do. The thing that I found frustrating was the lack of depth. The examples are fairly minimal, as are the explanations. If you don't understand

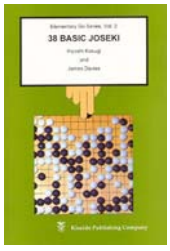
Bookshops

The following bookshops actively support ACCU (offering a post free service to UK members – if you ever have a problem with this, please let me know – I can only act on problems that you tell me about). We hope that you will give preference to them. If a bookshop in your area is willing to display ACCU publicity material or otherwise support ACCU, please let us know so they can be added to the list

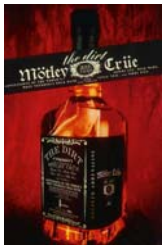
- **Holborn Books Ltd** (020 7831 0022)
  www.holbornbooks.co.uk
- **Blackwell's Bookshop**, Oxford (01865 792792)
  blackwells.extra@blackwell.co.uk

Desert Island Books (continued)

people wanting to bind tree branches together. These days, if I knew which title to order, I can buy through Amazon; before computers were widely available newsagents would give me strange looks when I asked for a copy of Byte magazine and was once told by a shop assistant 'We don't sell those kinds of books' when inquiring after a copy of a book about LaTeX (yes, I did use the correct technical pronounciation). I will have to leave it to those readers of CVu who know about Kinbaku to suggest which book I should take with me.



When my Man Friday turns up I will need a board game for the evenings. The game of Go is by far and away the best choice and probably the best book I have on the subject (not that I have many) is *38 Basic Joseki* (*Elementary Go Series*, Vol. 2) by Kosugi Kiyoshi and James Davies. The most expensive Go sets use pieces (small black or white ovaloids known as stones) made from shell, a material I will have ready access to.



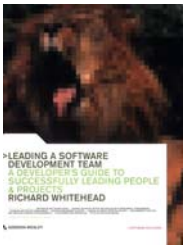
I have been asked to list at least one work of fiction, a classification to which many books on software engineering belong. The last non-technical book that I thought wonderful was *Motley Crue: The Dirt – Confessions of the World's Most Notorious Rock Band* by Tommy Lee, Vince Neil, Mick Mars and Nikki Sixx. Not a salacious book at all but a gripping, clear eyed and apparently brutally honest narative and self-analysis of the lives of members of a rock band.

If I had to chose a piece of music it would of course be Bach's *The Well-Tempered Clavier*. I prefer Glenn Gould's interpretation and have had the Sony remastered versions (using 20 bits, where did the other bits go or did they add to the original 16?) of book I and II (2 CDs in each of the two packages) as the only CDs in my car for the last five years.

Next issue: Lisa Crispin.

everything from the code and the short explanation, then you have to resort to digging around the Boost documentation, which somewhat defeats the purpose of reading a book. One thing that I must commend is the quality of the translation, better than some books written in the mother tongue.

Leading a Software Development Team: A developer's guide to successfully leading people & projects



By Richard Whitehead, published by Addison-Wesley, ISBN 978-0201675269

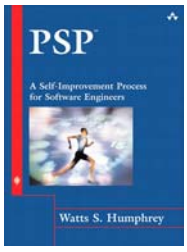
Reviewed by Paul Floyd

The target audience is the newly promoted 1st level manager. Whitehead's advice seems to be mostly anecdotal. The advantage to this is that his observations often cut to the quick, certainly more so than the more blurry observations made by larger statistical studies or theoretical methodologists. As an example, I thought the the remarks about 'Code-centred approach' (part 3 of Chapter 38) hit the mark very well. There are other occasions where I didn't agree with his views. His advice is more oriented towards achieving short-term goals. I would have liked to have seen more balance between that and the long term. Whitehead doesn't see much point in Non-Functional Requirements, whereas I feel that if you want long term quality goals like maintainability, then you're best specifying those objectives up front.

Another point that didn't match my experience is the idea that people get promoted to be managers because they are the best technically, and thus leading from a designer/architect role. From what I've seen, it's usually length of service and saying the right things to the level 2+ managers.

One last little quirk. A couple of times there is mention of subordinates wanting to play with the latest technology in order to pad out their CVs. That seems a strange consideration for deciding on how people should do their work.

A Self-Improvement Process for Software Engineers



By Watts S. Humphrey, published by Addison-Wesley, ISBN 0321305493

Reviewed by Paul Floyd

This is a book that I'm going to love to hate. On the one hand the cowboy coder in me just wants to shoot from the hip. And on the other hand I really would like to have some of that 1 defect per 1000 lines of code delivered on time stuff. Having read the book, I wonder if 'DRFS' might not have been an alternative title – *Development Reduced to Following Scripts*. As far as I can tell, this is all about 'getting it right'. Whose right though? If you're trying to do things that have never been done before, then I'm not sure that a mechanistic approach is the answer. Such a method would help with implementing the solution, but finding it? If you are far from the bleeding edge and your 'getting it right' is synonymous with 'not getting it wrong' then this may well be for you.

The other big problem that I had was with the examples. Clearly to get the best out of this book you need to do the examples. They seem to be designed to fit in with a 1-week course, lectures in the mornings and lab sessions in the afternoons. This means 3–4 hours for each example. I found that hard to fit into 'spare time', (which for me is usually in ½ hour to 1 hour chunks). That all assumes that you can even get to use the examples. You can download them from the Carnegie-Mellon SEI site. They are all based on Windows and Access. If you don't use Windows or don't have Access, then forget it. I didn't have Access when I started reading the book, but then I got a cheap copy through my employer. Not being familiar with it (and there are no explanations for complete neophytes) it was a bit of a struggle. No problem if you have an instructor at hand.

Where Good Ideas Come From – The Seven Patterns of Innovation

By Steven Johnson, published by Penguin, ISBN 978-0-141-03340-2

Reviewerd by Ian Bruntlett

Highly Recommended

I am not a creativity expert. But I do like to dabble :) For me this book has proved to be just as important as Edward de Bono's *Lateral Thinking* book. The core of this book is dedicated to 7 chapters discussing 7 patterns of innovation. Personally I have spent some time in a small R&D department and from what I have seen there, the 7 patterns of innovation are spot on. I'll review this book, part by part.

Introduction: Reef, City, Web. Examines extreme levels of creativity in these environments.

1 The Adjacent Possible discusses how innovation more often than not comes in small steps & rarely in great leaps.

2 Liquid Networks. How environment needa to be stable to support creativity – it must not be too volatile or too rigid.

3 The Slow Hunch again talks about the reality that leaps of imagination take place over a lengthy period of time.

4 Serendipity. This chapter suggests knowledge in one field of endeavour can lead to innovation in an unrelated field.

5 Error discusses how the role of heredity and changing environment shape a creature's evolution. For instance cloning requires less effort and resources than sexual reproduction. Sexual reproduction includes errors etc which allows a creature's offspring to vary and some of the variants will be rewarded by evolution.

6 Exaptation (the process by which features acquire functions for which they were not originally adapted or selected).

Borrowing a mature technology from an entirely different field and putting it into use solving a problem in another field.

7 Platforms. A Platform is a space that encourages innovation (e.g. 18th Century Coffee Houses, Home Brew Computing Club).

This leads to acts of creation that instead of opening a door to the 'adjacent possible' but results in the building of a new floor (GPS, Internet).

Conclusion – 'The Fourth Quadrant', studies innovations from 1400–2000 using a 4 quadrant classification system for innovations.

These innovations are described in detail in the Appendix . The categories are: 'Market' or 'Non-Market', 'Individual' or 'Networked'.

The numbered quadrants are:

1. Market/Individual – an innovator either alone or part of a small group that wants to profit directly from their innovations – e.g. Nylon, Revolver, Programmable Computer
2. Market/Networked – an unrelated collection of groups that wish to profit from a combined innovation – e.g. Aircraft, Personal Computer.
3. Non-market/Individual – an innovator either alone or part of a small group that shares the innovation without intending to profit from the innovation – e.g. Atomic Theory, World Wide Web.
4. Non-market/Networked. An unrelated collection of groups that collaborate on an innovation for a common cause – Modern Computer, Quantum Mechanics.

The chapter is called 'The Fourth Quadrant' because in our time – with the free transmission of ideas over the internet – the fourth quadrant work flourishes – and we all benefit.

Finally, in an attempt to encourage you to read this book, I must quote the final paragraph:

> The patterns are simple, but followed together, they make for a whole that is wiser than the sum of its parts. Go for a walk; cultivate hunches; write everything down, but keep your folders messy; embrace serendipity; make generative mistakes; take on multiple hobbies; frequent coffee-houses and other liquid networks; follow the links; let others build on your ideas; borrow, recycle, reinvent.

View From The Chair

Hubert Matthews
chair@accu.org

Education, education, education. It's a political catchphrase that was popular in 1997 but never really achieved what it might have done. Perhaps 2012 will be the year when it will mean something in the programming community. There are a number of constellations in the world of code that are aligned auspiciously so it will be interesting to see what happens. One such movement is Code Year [1] that is attempting to get people to learn to program using JavaScript. Their web site claims 392,473 people have signed up so far, including Michael Bloomberg [2]. I very much look forward to seeing our friend, bon viveur and purveyor of high-speed slides Mr John Lakos discuss value semantics with his uber-boss in the bar at the upcoming conference. Perhaps that might solve our sponsorship problems (then again, maybe not).

Another celestial indicator is the publishing of the Next Gen report [3] and the Royal Society's report on computing in schools [4]. The Next Gen report is looking to turn the UK into 'the world's leading talent hub for video games and visual effects' and the Royal Society is looking at 'the way forward for computing in UK schools'. Both bemoan the lack of programming talent and say that this starts in schools where 'ICT' focuses on office skills and not rigorous computer science, and that this is compounded by poor university courses. Something is obviously awry in the local water supply.

One more piece of this astronomical jigsaw is the 'ready real soon now' Raspberry Pi [5], a credit-card sized ARM-based computer with high quality graphics for $25 or $35. This is a fantastic idea and perhaps this will be the successor to the BBC Micro and the ZX Spectrum, machines that spawned a whole generation of programmers. Lots of members have told me they want to get their hands on one of these little beasties so we won't be short of people to help others with relevant knowledge.

The point here is that the pendulum seems to be swinging back. Programming as an activity is coming back into fashion after a number of years in the educational wilderness. The question for us in the ACCU is how we can help. I've spoken to a number of members who are very keen to get involved in such a project as they see this as a key aspect of what we should be doing. Should we offer programming advice via something like our existing members-only accu-prog-questions mailing list? Should we get involved with other similar organisations that want to support these initiatives? The range of possibilities for what to do is huge and we have limited time and resources (both at a member level and the ACCU as an organisation) so we need to be careful not to overstretch ourselves and also not to think (and talk) big but not deliver. However, I believe it is important that we catch this wave of enthusiasm and that we get involved in some way in this. I would therefore be eager to hear from members who wanted to volunteer some time for this as well as those with ideas of what we could and should do. Let's get our teeth into this, do something useful, help enthusiastic newcomers and make a difference.

References

[1] http://codeyear.com/
[2] http://www.bbc.co.uk/news/technology-16440126
[3] http://www.nesta.org.uk/events/assets/features/next_gen
[4] http://royalsociety.org/education/policy/computing-in-schools/
[5] http://www.raspberrypi.org/

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?