The magazine of the ACCU

www.accu.org

{cvu}

Volume 18 Issue 6 December 2006 £3

Features

Regulars

{cvu}
Volume 18 Issue 6
December 2006
ISSN 1354-3164
www.accu.org

Editor
Tim Penhey
cvu@accu.org

Contributors
Ryan Alexander, Mark Easterbrook, Lois Goldthwaite, Pete Goodliffe, Paul Grenyer, Thomas Guest, Jez Higgins, Andrew Marlow, Roger Orr, Tim Penhey

ACCU Chair
Jez Higgins
chair@accu.org

ACCU Secretary
Alan Bellingham
secretary@accu.org

ACCU Membership
David Hodge
membership@accu.org

ACCU Treasurer
Stewart Brodie
treasurer@accu.org

Advertising
Thaddeus Froggley
ads@accu.org

Cover Art
Pete Goodliffe

Repro/Print
Parchment (Oxford) Ltd

Distribution
Able Types (Oxford) Ltd

Design
Pete Goodliffe

accu

Do you get what you pay for?

If there is one constant in the world it is that things change. I have re-entered the world of permanent employment and it feels very strange. I am now working for Canonical, the company behind the Ubuntu Linux distribution. Canonical are the driving force behind many fun things, one of which is the bazaar distributed revision control system. That is something I'm going to have to write more about later.

Working for a company founded on open source ideals makes you think slightly differently about some things. I feel that there is the predominant opinion that with a lot of free software that you get what you pay for. Often the support and documentation of free software is somewhat lacking. Often the software is not easy to use, GUI polish is often missing and, in my experience, you get occasional crashes.

Much of this is understandable if you look at the people who work on open source software. Only a small proportion of the people working on open source software are paid to do so. Most programmers I know are more interested in getting something working than making it look pretty. Luckily these days there are user interface specialists who have become involved in the open source world and many of the applications are light years ahead of where they once were.

Is open source software really heading in the general direction of great successful projects like gcc, firefox, and open office – or are the successful projects just the anomalies? I'm hoping for the former.

TIM PENHEY,
EDITOR

The official magazine of the ACCU

The 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.

The 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 ACCU members – by programmers, for programmers – and have been contributed free of charge.

To find out more about the 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

COPY DATES

C Vu 19.1: 8th January 2007
C Vu 19.2: 1st March 2007

IN OVERLOAD

This month, you can read: 'Pooled Lists' by Christopher Baus, 'The Singleton in C++ – A force for Good?' by Alexander Nasonov, 'C++ Interface Classes – Strengthening Encapsulation' by Mark Radford and 'A Drop in Standards' by Paul Johnson.

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

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

By default, the copyright of all material published by ACCU is the exclusive property of the author. By submitting material to ACCU for publication, an author is, by default, assumed to have granted ACCU the right to publish and republish that material in any medium as they see fit. An author of an article or column (not a letter or a review of software or a book) may explicitly offer single (first serial) publication rights and thereby retain all other rights.

Except for licences granted to 1) Corporate Members to copy solely for internal distribution 2) members to copy source code for use on their own computers, no material can be copied from C Vu without written permission from the copyright holder.

Adventures in Autoconfiscation

Jez Higgins moves a project to GNU autotools.

Arabica is an XML toolkit written in C++[1]. It provides a SAX interface for streaming XML parsing, a DOM interface for in-memory XML processing, and an XPath engine for easy DOM access. In the next release or two, it will add an XSLT processor. Arabica supports `std::string`, `std::wstring` or pretty much any other crazy string class. The code itself is good, honest, standard C++, which my experience shows is highly portable. I've built Arabica on Windows using Visual C++, under Cygwin, on a variety of Linux flavours, FreeBSD, several types of Solaris, OS X, and GNU Darwin. It's quite a tidy package, and if you're working with XML in C++ you should consider it. That's what I think, anyway, but I did write it.

Damon, in Montreal, disagrees. On 22 February 2005 he wrote[2]:

> The C++ port of SAX (a set of standard JAVA API for XML parsing) Arabica is totally unusable, there are even syntax errors like namespace errors in the so-called stable release, besides it does not have any reference manual!

The following week he wrote[3]:

> It is an awful library with a bunch of syntax errors in its newest release (namespace error....), no documents available at all. Fail to compile it on Linux at all.

I didn't have any correspondence with Damon. I found his comments months later during a vanity searching session on Technorati[4]. In the nearly 8 years since Arabica's initial release, it's been my experience that people very rarely write to you about your software. When they download your code, it either works or it doesn't. If it works, they got on with what they were doing. If it doesn't, they may take a moment to think you're an idiot, then they fling it away and try something else. More often than not, that first point of failure is the build. If it doesn't build, it's fallen at the very first hurdle.

they may take a moment to think you're an idiot, then they fling it away and try something else

The Arabica distribution contains, currently, around 150 source files. Since Arabica is largely implemented as C++ templates, the majority of the files don't need to be compiled and built separately. You just include them into your code. Only a handful, under 20, need to be built into a shared or static library.

Why did Damon have such a hard time? Why didn't I?

Come with me. Come with me on a journey through time.

When I started on the code that would become Arabica, I was an angry man. I was having a very bad experience at work, with a rotten developer, who had handed over a horrible, verbose, bug-ridden piece of code. It was, allegedly, an XML parser. It read an XML wire-format and built a C++ object graph, what we now call deserialisation. At the time, I called it rubbish.[5]

I had argued that we shouldn't, absolutely shouldn't, build our own parser, but use one of free parsers that were, even then, already available. When I had this lump of code dropped on me, I wanted to demonstrate just how awful it was. I grabbed the Expat source[6] and got the build going on Windows. Next, I grabbed the recently released Java SAX interfaces[7], and ran though them search and replacing `String` for `std::string`. That done, I hooked up Expat, which is a C library that deals in `char*`, to my new SAX classes. It worked. No bugs. Not bad for an afternoon's work. I released the code as an afterthought. I didn't think it was of particular interest, but the code I'd based it on was freely available, and I needed something to put on my website.

Over the subsequent months, and then years, I continued to work on Arabica on and off. There was a new version of SAX, which I incorporated. Other C and C++ XML parsers were released [8][9][10], and I wrote SAX wrappers for them.

For most of that time, my primary development platform was Visual C++ 6 and then 7. Every now and again, I'd boot up a Linux box, refresh the Makefiles, and clean up the conformance errors GCC pointed out. It worked OK, after a fashion.

As the library grew, the build became increasingly fiddly. While Arabica provided bindings for Expat, libxml2, Xerces, and MSXML, you'd only want to build against one of those. That implies a certain amount of Makefile editing. I found out that some compiler/operating system combinations didn't support `std::wstring`, so parts of the build had to be conditionally excluded. C++ libraries have different levels of Standards conformance, and there are ambiguities in some places, so parts of my code have to be conditionally included to plug the gaps. Some platforms put things in different places, or expect certain types of files to have certain extensions[11], which needs more Makefile editing.

At the time Damon was discarding Arabica as completely unusable, my build notes were:

- Building Arabica isn't hard, but it can be a little fiddly.
- First, you will need to have at least one of the following parsers installed: expat, libxml, Xerces. If you're working on a Linux box, you probably have libxml or expat already installed. It's entirely possible to build in support for several parsers, but you'll probably only want one.
- Next you need to build the SAX library, configuring it for your choice of parser, or parsers.
- In an ideal world you'd just do `./configure` and be done with it. Unfortunately, at the moment the dark recesses of template meta-programming are as nothing to getting autoconf going. One day... So anyway, we have to resort to a little Makefile fiddling. What I'm going to describe is probably GNU Make specific, but for other Make variants you should be able to follow along OK.
- Choose your parser (or parsers) as detailed above.
- You'll need a relatively Standards compliant C++ compiler and library – gcc 3.x.y is okay, gcc 2.95.* will probably work if you use an alternative library such as STLPort.
- Untar the Arabica source.
- At the top level directory, you'll find a Makefile which builds everything. It uses the `-include` directive to pull in `Makefile.header`, which is where all the twiddly bits are.
- Pull up `Makefile.header` in your favourite editor. Most of it should be pretty obvious – defining CXX to point to your C++ compiler and so on. There are some examples in the distribution you can use as a base.

JEZ HIGGINS

Jez works in his attic, living the devil-may-care life of a freelance programmer. After work, he walks the dog. In April, he became ACCU Chair.
His website is http://www.jezuk.co.uk/

- The interesting `Makefile.header` macro is **`PARSER_CONFIG`**. **`PARSER_CONFIG`** controls which parsers Arabica will use, and also whether to compile in wide character support. For each parser you want to configure as **`-DUSE_parser`**. The choices are **`USE_EXPAT`**, **`USE_LIBXML2`** and **`USE_XERCES`**. If you don't need, or your platform+compiler doesn't support, wide characters (eg. Cygwin, gcc on Solaris) you'll also need to set **`-DARABICA_NO_WCHAR_T`**. For each parser you support, add the appropriate **`-lwhatever`** (**`-lexpat`**, **`-lxerces-c`**, **`-lxml2`**) to **`DYNAMIC_LIBS`**.
- Run **`make`**. **`libSAX`** should build, possibly with a number of warnings about preprocessor tokens, and finish up in `./bin`. If your parser's header files aren't installed in the usual places (`/usr/include`, `/usr/local/include` or whatever the default is for your platform), you'll have to edit **`INCS_DIRS`** in the `Makefile`. Once **`libSAX`** is built, everything else should build too.
- The supplied Makefiles work for me on using gcc on Suse Linux 7.3, Cygwin and Solaris 7. If you can supply a `Makefile.header` for a new platform+compiler, I'd be delighted to receive it.
- Once the SAX library is built, the DOM library is simplicity itself. You don't have to do anything! Arabica's DOM implementation is all headers files. If you want to use it, just include the appropriate parts, link the SAX library, and you're done.

You can see I had made some effort to ease this process. GNU Make supports an include mechanism, so I had moved all the platform specific pieces out into a separate Makefile fragment. This minimized the number of places that needed to be edited, but there was still a deal of manual intervention required. I supplied a number of platform specific versions, 6 at last count, but I didn't have regular access to all of the platforms in question. Note also the equivocation – "that should be it", "will probably work", "works for me". It wasn't reliable, and I knew it.

this cruddy, wobbly, unreliable build system that had accreted around the outside

It was a maintenance bother too. As I added more test and example programs, I had more Makefiles to maintain. When I added the XPath engine, which uses Boost Spirit, I received emails from people who didn't need XPath asking me how to leave out it out of the build, as their builds were now broken.

I had this code, code I knew was good and portable and useful, but I had this cruddy, wobbly, unreliable build system that had accreted around the outside. It was awkward for me, off-putting for other people. At least one person thought I was a useless idiot. Something had to change.

Out with Makefiles

I needed an alternative to my motley collection of Makefile bits and pieces. At the very least it had to meet the following criteria

- be able to find Arabica's prerequisites – at least an XML parser and optionally Boost
- identify whether **`wchar_t`** was supported
- detect platform specific file extensions
- track file dependencies
- be at least as easy to maintain as my existing setup
- stand a better than even chance of working on the random machine that somebody has just downloaded my code to

There are now many alternatives to Make. There are Ant[12], its Groovy derivative Gant[13], and its .NET-alike Nant[14]. There are Cons[15] and Scons[16]. There are Jam[17] and BJam[18]. There are Rake[19] and A-A-P[20] and a whole host more I'd not even heard of[21]. If you look at any of these tools, chances are there's at least a passing reference to how much better than Make it is.

I didn't consider any of them, not even for a moment.

```
$ wget http://somewhere/path/to/somelib.tar.gz
$ tar zxf somelib.tar.gz
$ cd somelib
$ ./configure
[... lots of output snipped ...]
$ make
[... lots more output snipped ...]
$ make install
[... a little bit more output, also snipped ...]
```

Figure 1

If you download some arbitrary program or library written in C or C++, from Sourceforge, Tigris, Savannah, or whereever, it won't, as rule, use any of those tools. Chances are pretty good that it won't need anything like the fiddling the Arabica did. You expect something like Figure 1.

Anything else violates the principle of least surprise by a considerable distance. There was only one choice for Arabica – GNU Autotools.

In With Makefile.am

The magic of the "./configure; make; make install" is provided by GNU Autotools. Autotools is actually three separate packages – **`autoconf`**, **`automake`**, and **`libtool`**. **`Autoconf`** creates portable and configurable packages, the configure script. **`Automake`** is a Makefile generator, used with **`autoconf`** to produce Makefiles based on what configure finds out about the system. **`Libtool`** is a set of shell scripts to build shared libraries in a generic fashion. In reality, you don't use one without the others. As far as I can tell Autotools is not an official name, but everybody knows that it means.

I found the whole process so dispiriting and confusing that I abandoned my efforts

You might have noticed a disparaging reference to configure in the build notes above. I've actually been here before. Six years ago, I attempted to convert Arabica as-was to use Autotools. Even armed with a hot off the press copy of New Riders' GNU Autoconf, Automake, and Libtool[25] – a book written by the primary Autotools maintainers – I made absolutely no progress at all. I found the whole process so dispiriting and confusing that I abandoned my efforts, subsequently consigning myself to years of creaking Makefiles and the contempt of Damon from Montreal.

Six years is a long time in programming. Despite my previous bad experience, I had no doubts that I would, in relatively short order, autoconfiscate[26] my project. Next time, I'll tell you how I did it. ■

Notes and References

[1] http://www.jezuk.co.uk/arabica
[2] http://damonli.blogspot.com/2005/02/track-separation-today_22.html
[3] http://damonli.blogspot.com/2005/02/track-separationace-spring-break.html
[4] http://www.technorati.com/
[5] I know everybody has "he was awful" war stories, but this was, genuinely, one of the worst experiences of my working life. I'm getting angry just thinking about it again.
[6] http://expat.sourceforge.net/ Expat was originally written by James Clark, a real XML big brain, and is widely used. It's still my XML parser of first resort.
[7] http://www.saxproject.org/. SAX describes a streaming XML parser interface. It was initially developed provide a common interface to XML parsers written in Java (as JDBC provides a common interface to databases), but there are now implementations in most languages.
[8] libxml, the GNOME XML parser, http://www.xmlsoft.org/

Effective Version Control #3

Pete Goodliffe concludes the series.

The previous two articles in this mini-series discussed what version control is, the basic usage of a version control system, and how to look after repositories and the code held within them. In this final instalment we'll look at some more advanced, but essential, parts of the version control story. We'll investigate good merging and branching practices, and how to use version control in your product release procedure.

Tags and branches

These two concepts are fundamental to managing files under revision control, and allow you to organise work in a logical, maintainable way. However many projects branch badly, or don't exploit branches enough – making their work harder and more dangerous. These items will show you how to employ branches and tags to your development advantage.

16. Separate development lines

Branches enable you to fork your development effort and work on different features simultaneously, without the development efforts interfering with one another. Once complete, each code branch can be merged back onto the mainline to synchronise the fork with its parent. This is an immensely powerful development tool.

Many common tasks are made much easier with branches. Use them for:

- Encapsulating major revisions of the source tree. For example: each feature should be developed on its own branch.
- Exploratory development work – the stuff you're not sure will work. Don't risk breaking the main development line: tinker on a branch and then merge down if the experiment is a success. You can also create multiple branches to test out different ways of implementing the same functionality; merge down the most successful attempt (a form of code natural selection?)
- Major changes that cut across a lot of the source tree and will take a while to complete, requiring many tests, and many individual check-ins to get right. Doing this work on a branch prevents other developers stalling for days on end with a broken code tree.

Here are some more contentious applications of branches that you might see employed in development teams. I recommend that you *don't* attempt these unless you definitely need to partition your developers' work. (This is yet another case of the simplicity rule – don't make complicated branch strategies when you don't need them.)

- Individual bug fixes. Open a branch to work on a bug fix, test the work, and then merge the branch down once the fault has been closed.
- In the extreme case assign a different branch for each individual developer; then there's no danger of their work breaking anyone else's. It's their responsibility to merge the mainline into their branch as required to keep their view of the world up-to-date. When there's something worthy of inclusion in the software release, they tell a code integrator who reviews the work, and then merges it into the mainline.

Branches an excellent organisation facility just waiting to be used. Don't be afraid of them. Don't pollute your main development line with unnecessary cruft that can be hived off into a branch.

17. Separate development from release lines

This is another good use of branches, but it's so important that it's elevated to it's own item.

As you approach a release deadline you must be *very* careful how you change the code. Simple modifications that would have been acceptable a month ago are no longer welcome. Every check-in must be justified and included in the release on merit, after careful review. Otherwise the integrity of the release code is in danger – one careless check-in could put you back by weeks.

However, deadlines are such that you've probably made a start on the next development task whilst a few developers are desperately trying to remove the last few bugs. Branches can keep all that disparate activity under control.

At a designated point create a *release branch*. This will contain the code that comprises your software release. Usually this is a line of development that is versioned: each build gets its own version number, and the source code for it is tagged at each build point, with a name reflecting this build number. (See item 20 for more on making releases.)

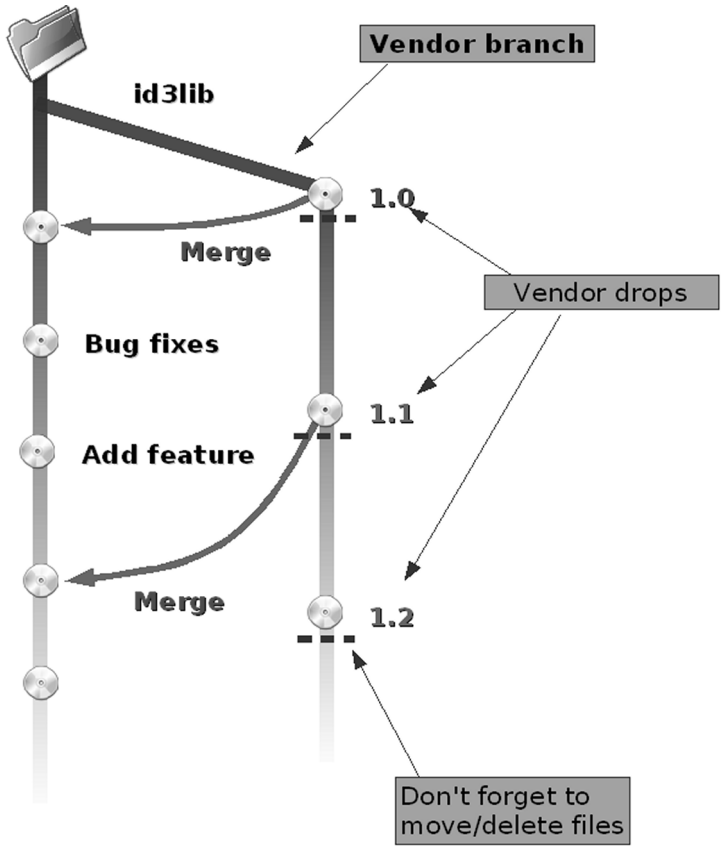
PETE GOODLIFFE

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@cthree.org

Adventures in Autoconfiscation (continued)

[9] Xerces-C is an Apache project initially donated by IBM, http://xml.apache.org/xerces-c/
[10] MSXML, the Microsoft XML parser packaged as a COM object, ubiquitous on Windows boxes, http://msdn.microsoft.com/xml
[11] Shared libraries, for example, generally have `.so` extensions. Under Cygwin, however, uses `.dll`, while OS X and other Darwin derivatives use `.dylib`.
[12] http://ant.apache.org/
[13] Gant is a build tool for scripting Ant tasks using Groovy instead of XML to specify the build logic. http://docs.codehaus.org/display/GROOVY/Gant
[14] http://nant.sourceforge.net/
[15] http://www.dsmit.com/cons/
[16] http://www.scons.org/
[17] http://www.perforce.com/jam/jam.html
[18] http://www.boost.org/tools/build/v1/build_system.htm
[19] Rake is Ruby's build tool, http://rake.rubyforge.org/
[20] http://www.a-a-p.org/
[21] There's a big (and undoubtedly incomplete) list at http://dmoz.org/Computers/Software/Build_Management/Make_Tools/
[22] http://sourceforge.net/ You knew this one didn't you?
[23] http://tigris.org/ ColabNet's Sourceforge-a-like
[24] http://savannah.gnu.org/ GNU Project's Sourceforge-a-like
[25] Updated text now available at http://sourceware.org/autobook/
[26] Autoconfiscation – the process of converting a project to use GNU Autotools

Figure 1



The release branch is kept relatively stable, and absolutely *no* development work will occur on it. The development continues down feature or bugfix branches. When a change is deemed fit for integration into the software release it is merged across into the release branch.

In this way you maintain a controlled release codebase, and can track the history of changes made between build revisions of any software release. At the same time you can attempt some more contentious bugfixes without compromising the software release. You can even be developing the next generation product at the same time.

18. Tag releases and milestones

Tags (or *labels*) are the version control equivalent of bookmarks, allowing you to mark important revisions of the software during its development, to easily get retrieve them later, and to move the bookmark when the world has changed.

It's good practice to use tags liberally. There are several scenarios that you should consider tags for:

- marking the code that comprises major software releases,
- marking code milestones against the development schedule (i.e. 'feature complete' or 'release candidate' points),
- personal code bookmarks to help a developer in their day-to-day work, and
- marking important points before/after complex operations. For example, it's good practice to tag the tree before a complex merge, and the again afterwards (see item 19).

Tags are identified by name, so choose a good naming scheme, distinguishing personal tags from releases, and from milestones. Personal tag names should ideally include the user name, date, and reason for creation. Milestone/release tag names should include the software release name and build number or milestone name.

19. Merge carefully

The reason you made a branch was to keep things compartmentalised, so only merge branches together when you definitely want the development lines to converge. Only merge down when a branch is shown to be in a good state, with code in an acceptable, working form. Otherwise you will pollute the clean mainline with partial work and broken code.

Perform the merge operation *very* carefully. Watch out for any warning messages, any errors, and any conflicts found during the merge, and don't commit your merge to the repository until you've verified its integrity. Some VCSs have more complex merge issues than others, so make sure that you understand all the implications of a merge in your environment. For example:

- How do directory changes (e.g. file creation) on a branch merge down to main – do you have to run a separate merge operation to pick those up?
- Does your VCS give you any support when you perform more than one merge from a branch to another? Do you have to remember that you've already merged versions 1-10, so that a second merge at release 20 doesn't try to import all the 1-10 changes again?
- Does the VCS record merge information for you to investigate later, or must you include merge information in your check-in message? (If you do have to include this merge information then be complete so that you can work out what you did later on. It's not a bad idea to include the entire merge command in the check-in message. At the very least record the source branch, the destination branch, the range of source revisions you're merging, and any other pertinent information).

It's good practice to tag the destination branch before and after the merge operation. This makes it easier to recover from a merge catastrophe, and also helps you to compare the effect of the merge operation on the branch.

19½. Reprise: Vendor branches

In item 6 we looked at managing third party code, by storing it in a *vendor branch*. This is a special use of the branching mechanism. We put the pristine third party code into its own branch – and never fiddle with it there. This third party code is usually a library of some sort, and has periodic releases by the vendor. From time to time we must upgrade to a new release. Each release of the code (known as a *vendor* drop) is committed into the vendor branch (and tagged for future reference). We then merge that branch into the main line of our development (see Figure 1).

On the main branch we can make any tweaks to the vendor's code (our own bug fixes, for example). The vendor branch remains as a virgin reference version of the vendor's code. Perhaps we will feed our bug fixes back to the vendor for inclusion in a subsequent release.

When a new drop of the vendor code arrives, we commit it into the vendor branch, and merge the differences on that branch down into the mainline. If the vendor has incorporated bugfixes that we already checked into the mainline ourselves, the merge operation should be simple.

There's nothing magic about the source code that goes into a product release

Making source code releases

There's nothing magic about the source code that goes into a product release. It's the same source code that you've been working on for months. However, the release *procedure* is stringent. These two items show you the right way to build release software from the repository.

20. Release from a virgin tree

An official software release is an important thing, and you can't risk any gremlin creeping in to disrupt production. It's easy to overlook changes you've made in your working copy that will affect how the software is produced. You might have modified source files, or altered some config to help you debug. It's easy to forget these things. The only way to be sure that you are building from a good source tree is to check out a fresh one before building.

You must tag the source files that comprise each major release so you can reconstruct it later, and it's important to do this in the right order. To make a software release, create the *release tag* in your working copy first. Then check out a virgin tree on that tag, and build there (presumably in a new session so that no environment variable, etc, can affect the integrity of the build).

This named tag allows you to repeatedly fetch the exact code that went into that release, and build an identical release at any time in the future – if you need to research a bug in old software, for example. Making this tag first ensures that your new working copy is exactly what you expect. Otherwise the repository's state-of-the-art might have changed before you check out a build tree.

Humans make mistakes. That's what we're genetically programmed to do.

21. Automate everything in sight

Humans make mistakes. That's what we're genetically programmed to do. The only mistakes computers make are ones that we program them to do. Our consolation is that they'll do it repeatedly, until we find the mistake and fix it. The best way to prevent silly human errors during complex check-outs, configurations, or build procedures is to take the whole problem out of the hands of fallible humans and give it to the reliable machine. Typos in commands and failure to follow exact procedures are common problems, and can cause all sorts of subtle errors.

Write scripts or macros to automate these activities. Document how to use them. The incidence of failure will decrease dramatically. There are other upsides to this strategy. Work is:

- faster (there's no need for all those commands and all those button pushes),
- easier (there's no need to remember all those commands and all those button pushes), and
- requires less documentation (you only need to document one command or button push).

Do as little as possible by hand. You'll guard against every developer error you could imagine (and more besides). By making others' lives as easy as possible, in the long run you'll be making your own life easy.

Miscellaneous

Finally, the unclassifiable law of version control…

22. Learn to script your version control

Don't rely too heavily on the convenience of GUI front-ends to your VCS. Whilst GUIs undoubtedly provide a productivity gain for simple operations, they do not provide you with the flexibility to meet the demands of item 21.

All good VCSs provide command line access to the repository. Learn to harness the power of the command line, and understand how to incorporate these commands in scripts. There are lots of uses for this:

- overnight builds, triggered automatically at a set time,
- unified checkout/build/deploy scripts,
- continuous integration (a term from extreme programming describing fully automated build/test cycles), and even
- extending your VCS with with extra facilities (several VCSs allow you to install trigger scripts on the server that get invoked during specific operations)

All's well that ends well

The very first item in this series "1. Use version control" is a prerequisite for predictable software development. Everything else we've looked at is just a refinement of that single piece of advice – "Use version control *well*".

There are three commonly held version control myths:

1. It's hard
2. It's not my job
3. I can live without it

All three are wrong. Very wrong. When version control is deployed well, when the right tool is used, and used sensibly, good version control practice is not hard at all; it's not rocket science. Version control is *everyone's* responsibility, not just some specialist IT infrastructure department. And you *cannot* live without it. Version control is absolutely essential.

I've covered a lot of ground in these three articles – there's a lot to be thinking about as you evaluate or deploy your development version control system. But this stuff has a profound effect on the quality of your development process, on how easy it is to work with your source code, and how easy it is to work with other developers.

If version control is your responsibility (clue: it is) then make sure that you understand each of the principles listed here, and put them all into practice. ■

Pete's book, Code Craft, is out now.
Check it out at: http://www.nostarch.com

Write for us!

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

What do you have to contribute?

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

If seeing your name in print isn't enough, every year we award prizes for the best published article in C Vu, in Overload, and by a newcomer.

Trouble with TCP

Mark Easterbrook tackles TCP timeouts.

This is a tale of being trapped between a rock and a hard place solving a legacy code problem for one of my clients. The application didn't work how the customer expected it to work, the company had not anticipated how the customer might use it, and everyone believed TCP was a reliable communications medium with guaranteed delivery of packets. Any solution had to be minimum change and a re-write was out of the question.

The application environment was a network of workstations running a command and control application. Each operator could choose how and what to view on their screen, but all the data came from a common database so that all operators saw any changes immediately. In addition, each operator could add local annotations to their own view, the electronic equivalent of drawing on the screen with an erasable marker pen! This local annotation system had an additional feature, the ability to send a copy of an annotation to the other workstations, but with a number of limitations:

- The data was only sent to currently active workstations.
- It was a data broadcast in that there was no way to select individual workstations in the send.
- It was a one-shot send: Subsequent changes were not propagated.
- Any received data overwrote the local copy, regardless of which version was the latest.
- If a receiving workstation did not already have a copy of the annotation, it was not displayed unless the operator chose to see it.

The customer used this feature a lot, and soon found that sometimes the data would sometimes take hours to reach other workstations. As the data could be updated by multiple operators this was unacceptable. Of course, it was not possible to tell the customer they were using the feature incorrectly – they were already committed to it – if only we could get it to work reliably for them.

At first, the shipping code did not make any sense, and it was only after working forward through the source code control archive from the original over 10 years ago, reconstructing the development path of the feature, that it was possible have any chance of understanding it.

At the implementation level, the data was put into a spool directory which the transmit process would poll regularly. For each file, it would open a TCP connection to a host, and if successful, send the file and close the connection before moving on to the next host. As long as at least one host was available the file was removed from the spool directory. This probably worked fine when tested in the lab. It also worked adequately in the field.

Sometime later, and maybe nothing to do with this particular module, the open TCP connection library function was enhanced to include a 0.5 second timeout using the `setitimer()` function which takes 2 values: seconds (0) and microseconds (500000). This meant that when some machines were unreachable, no process iterating through all remote hosts would seem to hang (at least not long enough for the user to notice).

In the fullness of time as external network access became cheaper one of the customers started using a WAN instead of a LAN, then complained that the data transfer to remote machines was unreliable. The problem was that distant machines were taking more than 0.5 seconds to reply to a connect request. An obvious fix was to increase the timeout and this was done by adding a zero to the timeout. Although this made the problem go away, nobody had noticed that the intended change had in fact removed the timeout completely [1] at the application level. The code now used the connect timeout in the TCP stack which can be anything from 20 seconds to tens of minutes depending on the system.

MARK EASTERBROOK

Mark is a software developer working with embedded systems, high performance/reliability/availability systems, operating systems and legacy code. He can be contacted at mark@easterbrook.org.uk

Listing 1

```
static void NoReply(void) {}

  signal(SIGALRM, NoReply);
  alarm(5);
  write(fd, request, request_size);
  if (read(fd, reply, MAX_REPLY) < 0) {
    if (errno == EINTR)
    ...
  alarm(0);
  signal(SIGALRM, SIG_DFL);
  ...
```

Testing with the new longer connect timeout showed significant delays when there were unreachable hosts so yet another fix was needed. The answer at the time was to cache the open connections (I'm not sure what the question was), so the sending application would open connections to all available remote hosts and hold an array of open file descriptor so when it had a file to transmit it just need to call send and not open. Every two minutes it would close and re-open all the file descriptors so that the list didn't get too stale. Again, this probably worked fine when tested in the lab, but isn't resilient in a real network.

As seasoned network programmers know, it is very difficult to detect a failed connection without sending something, which is why many systems send heartbeats to each other. The other critical piece of information that the last fix didn't take into account is that the timeout on an open TCP connection is long, very long. In this case it was 20 minutes. The introduction of caching open connections without actually checking them for failure meant that if a remote host hard-failed (i.e. no TCP close requests were received), the process would hang for 20 minutes. The most common cause of such a hard fail is a router or WAN link outage, resulting in multiple remote connections failing, and thus multiple 20 minute timeouts. Such outages are not very common (nor are they rare), but they shouldn't take out local connectivity as well as remote!

About this time I became involved. As this was fragile legacy code the rules of engagement were minimum change and minimum risk, so a rewrite using multiple processes or a similar solution was out of the question.

Attempt 1: Add a timeout to the send

The code in Listing 1 sets an interval timer for 5 seconds and an empty signal handler for the alarm that will be generated. The read from the TCP socket will therefore be interrupted after 5 seconds if no data is received. The interrupted read will exit and set `errno` appropriately. That is the theory. In practice, it doesn't do this. In practice, it depends on which flavour of Unix you are running, which of course makes any solution non-portable.

The problem the Unix developers had was that any interrupt will cause a `read` to fail, which means that every `read` needs to handle interrupts and restart the `read`, for example:

```
while (read(fd, databuf, MAX_READ) < 0
  && errno == EINTR);
```

Even if this had been introduced from day one, there would still have been programmers surprised that their `read` calls failed in the presence of

interrupts. To introduce the behaviour later would cause code to fail after an OS upgrade. Different flavours of Unix addressed the problem in various ways:

- 4.2BSD the read was restarted after an interrupt. This means the above code cannot possibly work.
- 4.3BSD realised that programmers needed a way to interrupt the read, so introduced an **SA_RESTART** flag. This applies to the interrupt handler (via **sigaction**) and therefore does not work if you don't have access to the interrupt handler. (On other Unix variants there is similar behaviour using **SV_INTERRUPT)**.
- Solaris 9 does not seem to support any way of preventing a restart. This is one of the platforms on which I needed the code to work.

Attempt 2: Turn off non-blocking

Sockets have the option to run in non-blocking mode by setting **O_NONBLOCK** (or **O_NDELAY**). This causes the **read** to exit immediately, returning **0** if no data is available to read. A simple addition to my interrupt handler is all that is needed:

```
static void NoReply(void) {
    fcntl(fd, F_SETFL, O_NONBLOCK);
}
```

This worked on the first system tested – Solaris – but failed on the second – HP-UX. It seems that some kernels don't allow the blocking flag to be modified while a read is in progress. This was also a less than satisfactory solution because the interrupt handler requires access to the file descriptor, forcing it from function scope to file scope and forcing a unique interrupt handler for each instance of the **read** call.

Attempt 3: Close the socket

If the socket has failed it does not really matter what happens to it, so I tried a sledgehammer approach and just added a close socket call. The socket still needs to be non-blocking to cause the **close** to return immediately:

```
static void NoReply(void) {
    fcntl(fd, F_SETFL, O_NONBLOCK);
    fclose(fd);
}
```

This worked on Solaris, but not on HP-UX, probably for the same underlying reason why the non-blocking version didn't.

Attempt 4: The C version of try...catch

The solutions so far, as well as being non-portable and not working on HP-UX, didn't feel right. Expanding the scope of a variable was definitely a sticking plaster solution and probably would have come back to bite at a later date.

The underlying problem is getting cleanly out of a block of code and tiding

C also has try and catch, it is just well disguised

up when a unrecoverable error condition occurs. Some other languages would be looking at a **try**…**catch** block to try and solve it. Fortunately, C also has **try** and **catch**, it is just well disguised and not many C programmers know about it – see Listing 2.

If you think of the **if sigsetjmp** as the **try**, and the matching **else** as the **catch**, it is not dissimilar to a real **try..catch** block, albeit an uncouth stack reset rather than a stack unwind. The **sigsetjmp** saves the stack context and returns zero to indicate it has been called normally. The code then sets up the alarm and signal handler as before. If the alarm fires the interrupt handler performs a **siglongjmp** to restore the stack context

Listing 2

```
static void NoReply(void) {
    fcntl(fd, F_SETFL, O_NONBLOCK);
    siglongjmp(jmp_env, 1);
}

  if (sigsetjmp(jmp_env, 0) == 0) {
    signal(SIGALRM, NoReply);
    alarm(5);
    write(fd, request, request_size);
    if (read(fd, reply, MAX_REPLY) < 0) {
      ...
    }
    alarm(0);
  }
  else {
        ... (handle timeout)...
```

to that of the **sigsetjmp** call, which returns a non-zero value to indicate a jump entry, and takes the **else** clause thus avoiding the troublesome **read** call.

The **sigsetjmp** is a powerful error handling idiom, but needs to be used with care:

- The **siglongjmp** call must only occur when the main code is within the "**if sigsetjmp**" context so the jump can only occur from an inner block to a containing block. In this case this means making sure that the signal handler is disabled as soon as the **try**…**catch** section is exited.
- The stack is simply reset, so the "**catch**" block needs to tidy up, particularly if resources have been obtained. There is no stack unwinding to assist like in C++, so all allocation and deallocation must be carefully designed and checked for leaks.
- **Seglongjmp** smells like **goto**, so will fall foul of the anti-**goto** crowd. As with legitimate uses of **goto**, the alternatives are even worse, so it is worth the effort the justify it.

Alternative solutions

The constraints given at the beginning prevented any design time solutions to the problem, but it is worth quickly looking at how the problem could have been solved if I had had a free hand, in no particular order:

- Use **select()**. This allows the code to wait for one or more file descriptors with a timeout.
- Use non-blocking I/O and poll for replies.
- Use one process (or thread, if you like to live dangerously) for each connection.
- Use uni-directional messages (fire and forget). As this system didn't perform re-transmits, checking for a reply was a waste.
- Use UDP uni-directional datagrams.

Programming notes

I have used **signal** in the above examples, but the API to set a signal handler varies between platforms:

- signal (traditional System V)
- sigvec (BSD 4.x)
- segset (SVR3)
- sigaction (POSIX)

The **setjmp-longjmp** code was taken from:
http://www.developerweb.net/sock-faq/detail.php?id=202

Note

[1] 5000000 modulus 1000000 = zero, and zero means no timeout!

Library Vendors and the non-Existent C++ ABI

Andrew Marlow discusses possibilities around a missing standard.

There is no standard Application Binary Interface (ABI) for C++. The consequences when linking C++ binaries from dis-similar compilation environments are typically fatal link time errors. As a simple example, consider the function:

```
void doSomething(int flag);
```

Due to function overloading the name the compiler generates as the external symbol is slightly different from **doSomething**. The actual name might be **__Z11doSomethingi**. Another compiler for the same operating system might produce a different name, say, **?doSomething@@YAXH@Z**. This is done to provide a way of encoding additional information about the name of a function, structure, class or another datatype in order to pass more semantic information from the compilers to linkers. This process of changing the name of symbols with external linkage is known as name mangling. In our example, if an object file refers to the routine as **__Z11doSomethingi** but the definition is in an object file that refers to it as **?doSomething@@YAXH@Z** then the link will fail with a missing symbol for **__Z11doSomethingi**, due to name mangling differences between the two object files. This can happen if the first object file is produced by a compiler with a different ABI than the compiler that produced the second object file. Name mangling is the simplest example of ABI issues but there are several more ways in which the conventions for object file layout may differ.

language standards do not specify the kind of details that allow binary interoperability between dissimilar environments

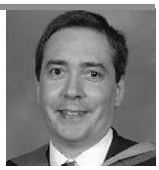
This normally means that a vendor of a commercial C++ library has to choose from one of the following three options:

1. Hand over the source code to the users.
2. Offer a C API alternative that does not contain any C++ (this works due to the de facto C ABI, mentioned later).
3. Support every customer configuration that there is.

These may all seem like drastic choices. This article explains why there is no alternative at present. The aim of pointing out these problems is for the following beneficial outcomes:

1. That developers become more aware of the dangers of using commercial specialist C++ libraries for which the source code is not available.
2. That pressure is placed on the C++ standards committee to get them to at least consider standardizing certain aspects of the C++ ABI (e.g. the name mangling convention) to mitigate some of the dangers and promote interoperability between closed-source libraries.
3. To get the developers of commercial closed-source C++ libraries to consider offering a purely C-based interface until these issues are resolved. This might seem like an unnecessary constraint on the library developers but it makes the libraries available to more users, some of which might otherwise not be able to purchase the library licenses.
4. To increase awareness of the developers of commercial closed-source C++ libraries in the benefits of exercising restraint in the use of certain C++ language features. It can minimize the trouble users will have trying to combine such libraries into an executable of their own. This also makes such libraries available to a larger number of users.

ANDREW MARLOW



Andrew has been programming for over 20 years, mainly in C and C++ on Unix. He began on machines that used punched cards and paper tape. Website: http://www.andrewpetermarlow.co.uk.

What is the C++ ABI?

Definition

According to Wikipedia, an Application binary interface (ABI) describes the low-level interface between an application program and the operating system, between an application and its libraries, or between component parts of the application. An ABI differs from an application programming interface (API) in that an API defines the interface between source code and libraries, so that the same source code will compile on any system supporting that API, whereas an ABI allows compiled object code to function without changes on any system using a compatible ABI.

For developers this means that when they try to combine libraries from more than one source (e.g. a home-developed library with a commercial one), the concerns are that it links successfully and that it runs successfully (assuming for the sake of this article that the code has no bugs).

Standardization efforts

In general, language standards do not specify the kind of details that allow binary interoperability between dissimilar environments. Even C has no such official standard. What is needed is a common object file format. This common format will differ from operating system to operating system, but on a given operating system it needs to be the same no matter what compiler is used, and no matter how the programming language evolves over the years. This is no trivial matter, even with a straightforward language like C, but over the years the standard object file format known as Executable and Linking Format (ELF) evolved for System V Unix (including Linux). On Microsoft Windows a similar standard emerged.

ELF means that two or more object files of C code from dissimilar compilation environments (say, two libraries produced by different vendors) can be combined into a single executable with no trouble. The vendors just need to make their libraries available for the same operating system (obviously, the machines instructions must be for the same instruction set). The trouble is that C++ has certain object layout and external symbol considerations that mean conformance to ELF is not enough. This is an area in which people on the standards committee feel that changes to the C++ language standard are not required. It is felt to be an implementation issue. After all, the details of ELF are not covered by the C standard and ELF is only a de facto standard.

Several compiler vendors have independently decided that their own compiler should define an ABI that is particular to their compiler, but this is hardly a standard. The Sun Solaris compiler ABI has stabilized as of Studio 7 (ref 1). The Free Standards Group (FSG) has created the Linux Standards Base (LSB) in an attempt to deal with the issue (ref 2). The ABI of the GNU C++ compiler was formalized starting with version 3 (but unfortunately changed at versions 3.3 and 3.4, causing problems for the LSB effort). CodeSourcery, a company based in California, is attemping to standardize on a C++ ABI for use in a GNU environment (ref 4). Other companies such as Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI are working with CodeSourcery on this. However, the effort currently does not include the big players Microsoft and Sun and only applies to the GNU environment. Furthermore, it does contain some processor-specific material for the Itanium 64 bit. The FSG started talking to ISO about ABI standardization in 2003 which culminated at the end of 2005 in ISO 23360, which standardizes the C++ ABI as part of a Linux standardization effort. But this is particular to linux. Whilst laudable, there is a lot more work to do before two libraries produced by different vendors can be combined into a single executable with no trouble on a commercial Unix system or on Microsoft Windows.

Common object file format

There are three main types of object files.

1. A relocatable file holds code and data suitable for linking with other object files to create an executable or a shared object file.
2. An executable file holds a program suitable for execution; the file specifies how the program's process image will be created.
3. A shared object file holds code and data suitable for linking in two contexts. First, the link editor may process it with other relocatable and shared object files to create another object file. Second, the dynamic linker combines it with an executable file and other shared objects to create a process image.

C development worked within this framework. The vendors of C libraries just shipped the header files and library archives (static and/or shared). The entrypoints mentioned as prototypes in the headers were resolved by linking into the associated libraries. Care was taken to ensure that the following aspects were taken into account in ELF and the behaviour of all C compilers for a given operating system:

- size and layout of predefined types (char, int, double, etc.)
- layout of compound types (arrays and structs)
- external (linker-visible) spelling of programmer-defined names
- machine-code function-calling sequence
- stack layout
- register usage.

C++ object format issues

an unstable ABI is incompatible with the safe use of shared libraries

C++ introduces some extra considerations. Unless these are taken into account and resolved, C++ libraries from different development environments even with the same operating system might not be successfully combined into an executable. It may fail at linktime. Even worse, there is the possibility that it may fail at runtime.

One of the reasons for the lack of a standard ABI is that as C++ evolved over the years, the ABI used by a compiler often needed to change in order to support new or changing language features. Programmers expected to recompile all their binaries with every compiler release. An unstable ABI is incompatible with the safe use of shared libraries, and is a nightmare for library and middleware vendors.

Here is a list of the C++ ABI issues that may cause problems:

- external spelling of names (name mangling)
- layout of hierarchical class objects, i.e., base classes, virtual base classes
- layout of pointer-to-member
- passing of hidden function parameters (e.g. this)
- how to call a virtual function
  - vtable contents and layout

there is a lot more work to do before two libraries produced by different vendors can be combined into a single executable with no trouble

  - the location in objects of pointers to vtables
  - finding adjustment for the this pointer
- finding base-class offsets
- calling a function via pointer-to-member
- managing template instances
- construction and destruction of static objects
- calling functions implemented in other languages and honouring their calling conventions.
- throwing and catching exceptions
- some details of the standard library
  - implementation-defined details
  - typeinfo and run-time type information (RTTI)
  - inline function access to members

ABI mismatch problems

Link time failures

The most common problem is where the program fails to link due to missing symbols. The classic case is where the naming mangling is different, but it can also arise due to the use of certain languages features in one environment that are not employed in the other. An example is RunTime Type Information (RTTI). An application that does not use RTTI itself and has the feature turned off (it is sometimes off by default) may fail to link with a library that makes use of RTTI. Another reason is where the different environments do not agree on the template instantiation mechanism. When libraries do not contain the template code for templates it instantiates the linker may fail to find these symbols at linktime.

Sometimes it may take a while before a failure to link is uncovered in the library. It may depend in which library facilities are called upon by the application.

Another problem that is a bit more unusual but still manifests as fatal link time errors is when the library has employed C++ internally but is not sold as a C++ library. This occasionally happens with C libraries. The company that produces the library are unaware of these issues and allow the library developers to do the majority of the implementation in C++ which gets a thin C wrapper added at the end. The user of such a library might be unaware that the library contains references to C++ and might try to link it with a C or FORTRAN linker, say, depending on what programming language they are using. This has been done by at least one software house employed by a Stock Exchange. The software house was told to provide an API for the price feed that can be called by C programmers. They produced one where the functions have C linkage but much of the code is written in C++.

Another subtle variation on this is that some vendors produce a C library but compile it with a C++ compiler, treating C++ as a better C. The external functions are all declared to have C linkage and the vendor is unaware that their library contains any references to C++. An example of a library that does this is the Mark 7 NAG C library (a maths library from the Numerical Algorithms Group). This contains references to C++ because all programs that contain C++ will need any static objects such as `std::cout` initializing before subroutine main is executed. This is done by the C++ runtime library and requires linking with the C++ linker. In the case of C libraries containing C++ implementation either directly or indirectly, I refer to this as "the C++ poison pill".

Runtime failures

Not all of these ABI issues cause linktime failure. For example, if a closed-source commercial library from one environment is linked into an application from another environment, and the environments differ on how exceptions are caught and thrown, the library may throw an exception that the application cannot catch successfully, even though the code in both places is flawless. Another example is where differences in compiler version cause differences in internal STL implementation details. Differences in object layout might mean a failure at runtime. This also happened to me only recently, where an application that used a library built with a recent version of g++ was accidentally linked with an older version of g++. This caused the program to core dump when it executed the default constructor for an ofstream whose internal structure had changed in the area of C locales.

The points mentioned above are often not understood by the vendors of commercial closed-source C++ libraries, or the development teams of those vendors. This means that even when the problem is understood by a potential user of the library, they may be told wrong information by the vendor. For example, the vendor might claim that if the user employs the C API in his code then no references will be made by the library to any C++.

some vendors produce a C library but compile it with a C++ compiler, treating C++ as a better C

What's a customer to do?

What does a customer do when they need to purchase a commercial C++ library from some vendor but the vendors C++ environment is different from that of the customers? (assuming the same operating system and same compiler vendor, just different versions of the compiler). The action really has to be taken by the library vendor but very often the challenge is getting the vendor to understand the issue and then getting them to agree to do something about it.

Vendor choices

Here are the choices that need to be offered to the vendor:

1. Hand over the source code to the users.
2. Offer a C API alternative that does not contain any C++.
3. Support every customer configuration that there is.

These are all drastic choices. They are not the only choices. For example, a vendor may offer an interface via a CORBA service where the server runs on some remote site. The vendor provides the Interoperable Object Reference (IOR) for their service and the Interface Definition Language (IDL) that describes the offered service. Then the user of this service doesn't even have to use C++ if they don't want to. But solutions like this are most unusual. Most vendors assume that their product will have to be linked into the executable. It is almost certain that a vendor of a commercial C++ library will choose from one of the three possibilities listed above when it comes to resolving C++ ABI issues. Another possibility, which only applies to Microsoft platforms, is to use COM. This works for the same reasons that CORBA works, but unlike CORBA is tied to the Microsoft environment.

Distribution of the source code Most vendors will consider that the first option could never be an option for them. They would not even offer their source via some escrow agreement in the event they went bust. They consider that it is only by keeping the code secret that they can sell the library at all. This is a great pity because there are several examples of companies that have taken this option with no ill effect to their revenue at all. For example, there is Troll Tech's QT library for GUI development. This library has a license that allows the source code to be distributed. The code is protected from being appropriated by another commercial company by the protections afforded by the GPL (the GNU project's General Public License). Another alternative approach which allows the source code to be distributed is to charge primarily (or exclusively) for support. This is the approach taken by Riverace in their commercial support of the open source C++ project ACE (Adaptive Communications Environment). Another example is ZeroC, which produces the open source product ICE (Internet Communications Engine). This is released under the GPL but a commercial license is available on request for users of ICE on commercial projects that do not wish to be bound by the terms of the GPL.

Perhaps more companies engaged in the development of commercial C++ libraries could be encouraged to follow the example of companies like Trolltech, Riverace or ZeroC. It is my belief that this is the best choice out of the three. It means their are no environmental mismatch problems and the source code is protected by copyright law, one of the most powerful kinds of law there is. However, knowledge of copyright law and how the protection it affords can be harnessed by use of appropriate source code licenses is outside the experience of most commercial companies. Even their legal departments seem to have little knowledge in this area.

Offer a C API The second option is seldom found and even when it is, it tends to be because the C API was developed first. A commercial example is the MQSeries library from IBM. It is rare to find a C++ library that was developed for C++ users but was done so by first implementing a core in C then wrapping C++ on top. An example of a small software package that does this is the sourceforge Cyclic Logs project (ref 3). One of the reasons why this is rarely done is that in non-trivial projects the developers wish to use the power of C++ and do not want to be constrained by using its less expressive relative, C. Care must be taken when discussing this option with a vendor. The vendor might not realize that a C API that involves any of the library code being compiled with C++ is still likely to cause problems.

Matching the customer's configuration The last option might seem to the most unpleasant as the amount of work involved increases in proportion to the number of configurations all the vendor's customers have, but in practice this seems to be the approach that most vendors take.

The most common problem this causes is that the library might not be available at all depending on what operating system and compiler is in use. For example, some commercial libraries might only be available on very old versions of Solaris. This would tie the developer to that version. If this is not acceptable the developer might have to face the fact that the library cannot be used on their project. I recently came across a situation where a vendor was using Visual Studio version 6 to compile a C++ library and a potential customer was using the Visual Studio 2005 since they also needed to use .NET. The customer pointed out that the ABI issue meant they might not be able to use the vendors library. The problem also affects some projects that use the GNU C++ compiler. This sometimes happens with projects that make use of Open Source projects. The Open Source projects typically start development with the GNU C++ compiler, because it allows the developers to use a relatively modern dialect of C++, compared to that supported by many commercial compilers. It is also free (as in free beer). But the project then wishes to use a closed-source commercial C++ library. At this point they may find out that the vendor does not support the GNU compiler.

The approach of matching the customers configuration is often used where the library in question is the C++ standard runtime library with associated STL, from the vendor of a commercial C++ compiler. The runtime library and the STL implementation tend to be tightly bound for commercial compilers, as the STL code tends to be expressed in terms of the runtime library. There are other examples of companies that take this approach for other kinds of library. For example, Rogue Wave have sold the commercial C++ foundation library Tools++ for many years. Users of Rogue Wave software may have experienced several of the issues described in this article as a result of unexpected differences between their development environment and the one used by Rogue Wave that produced the copy of the Rogue Wave library in use. These difficulties tend to get ironed out over time as the differences are enumerated and eventually the developers get the library that does exactly match their configuration. Matching the configuration can prove to be very tricky in some cases; in extreme cases it means that not only must the operating system be the same, at the same version, but both operating systems must have the same patches applied in the same patch order, and the compiler must be the same, the same compiler version and the same compiler options used, not only options that control optimization/debug but the use of exceptions, RTTI and multi-threading. For example, I experienced a problem once when using a library that required the developer use a special compiler option that worked around a bug to do with the lifetime of temporaries in the Solaris compiler.

Other developer options

Use lowest common denominator C++ If the developer cannot get the vendor to do any of the above, then the developer may be forced to adopt an environment that is as close to the vendor's as possible. This might mean using an older version of the operating system and/or older compilers. The use of older compilers could easily mean the developers must forgo using a more modern dialect of C++. This might restrict them in the area of templates and namespaces for example. Some parts of the STL that require a lot of template support might no longer be available to them. If the developer can put up with this then this might allow use of the commercial library.

Use Service Oriented Architecture The developer might make the library available as a service using CORBA or DCOM. An executable that offers the service could be linked on a machine that matches what the vendor says the customer must have. The customer can then use the facility via the CORBA/DCOM service in the environment of their choice.

Influence compiler writers Until relatively recently the ABI issue was not even being tackled by the developers of the same compiler. This is now changing. For example, both the GNU compiler and Suns' Studio compilers are now trying to adhere to ABIs developed by the team responsible for the compiler. This is an attempt to minimize ABI issues provided one sticks to the same compiler. However, not every commercial compiler is tackling the issue. Perhaps developers could try to put pressure on the companies that produce these other compilers to follow the example of GNU and Sun.

Influence the C++ standard C has a standard ABI on UNIX and Microsoft Windows, but it is only a de facto one. Nonetheless this has worked extremely well. However, there is no standard for a C++ ABI. Some people consider that this is not an issue for the standards committee; they say this is an issue for the vendors. If the vendors got together and a de facto standard emerged then it would be similar to what evolved in the world of C. This is not much help to the embattled developer.

It is proving hard enough for the same group of compiler writers to arrive at a consistent version of the ABI for the same compiler. The chances for complete interoperability between different versions of the compiler seem slim indeed. Perhaps a compromise would be for the standards committee to set a standard for name mangling. This is usually the first problem the developer encounters when their application fails to link. Sometimes when the developer and library environments are close (e.g. same operating system version, compiler produced by the same vendor buPowert at slightly different versions), same general dialect and features of C++ in use, a common name mangling convention might be enough to get the application to link successfully. At the very least it should serve to get those compiler developers that have not yet done so to confront the issue.

an unstable ABI is incompatible with the safe use of shared libraries

Conclusion

The developers and vendors of commercial C++ libraries need to be aware that not supplying the customer with the source code means they will have to support all the popular customer compiler configurations. In the past this meant offering libraries built with the current version of the commercial compilers for Microsoft Windows and commercial Unixes. Now, the GNU compiler needs to be included since it is becoming more widely used in commercial software projects and there are efforts at GNU ABI standardization. Potential customers will have to be aware that if they do not have a compiler configuration that is extremely common in a commercial environment then they may not be able to obtain a particular C++ library that will work for them. Sometimes this means they might be forced not to use the GNU compiler, or they might be forced to use a different version of the commercial compiler they already have. Sometimes they may get conflicting compiler version requirements if they are using more than one commercial C++ library.

The effort at CodeSourcery shows that a number of vendors consider the issue to be important enough to standardize on for at least the GNU environment. This gives scope for some optimism. The ISO standard 23360, whilst not being specifically for C++, also shows that there is some hope, at least for Linux. However, these are only recent developments. They only apply in limited areas and, even there, require the compiler to conform to the standard, something it typically takes some time to achieve. In the meantime, it is not safe for developers to assume that library vendors are aware of the issues. Neither should the developers assume any commercial libraries they might to use will be available for their particular compilation environment. These risks need to be identified very early on in a project that chooses to use commercial C++ libraries. It forms an essential part of any business case proposal for the purchase of a license to use such libraries.

References

[1] http://developers.sun.com/sunstudio/articles/CC_abi/CC_abi_content.html

[2] http://lists.freestandards.org/pipermail/lsb-discuss/2004-August/002162.html

[3] http://cycliclogs.sourceforge.net

[4] http://www.codesourcery.com/cxx-abi/abi.html
The ABI specification for the LSB

[5] http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=43781&scopelist=PROGRAMME Linux Standard Base Core Specification 2.0.1

String Literals and Regular Expressions

Thomas Guest wrestles with regex.

According to the *Draft Technical Report on C++ Library Extensions* [1] (more commonly known as TR1) regular expressions are making their way into the C++ standard library. Actually, Boost [2] users have had a regular expression library [3] for a while now. The library is beautifully designed and easy to use but is let down by the limitations of string literals.

String literals

Let's go back to basics and examine a C++ string literal:

```
char const * s = "string literal";
```

Here, the string literal comprises the sequence of characters `s`, `t`, ... `l`. The double-quotes (`"`) serve to delimit the contents of the string.

All's fine until we need a double-quote inside the string:

```
char const * s
  = "The "x" in C++0x will probably be 9";
```

This line of code gives a compilation error:

```
error: expected ',' or ';' before "x"
```

since the first internal double-quote closed the string. But how can we include a double-quote without closing the string?

Escape sequences

Here's how: the backslash is treated as an *escape character*. That is to say, normal interpretation of the string is suspended for a while – in this case for a single character – allowing us to write:

```
char const * s
  = "The \"x\" in C++0x will probably be 9";
```

Here, the internal double-quotes have been *escaped*, so they don't close the string literal but are in fact interpreted as double-quote characters within the string itself. Yes, it's confusing.

Literal backslashes

Now, if the backslash takes on a special meaning, how are we to insert a literal backslash into the string? Simple – we must escape that too:

```
char const * s
  = "A backslash \\ starts an escape sequence";
```

Here, despite first appearances, the string contains just a *single* backslash character. We did say it was confusing! Which leads us on to ...

Regular expressions

Put string literals aside for now. We're going to talk about regular expressions (let's call them regexes from now on). Regexes are used to find and match patterns in blocks of text. Like string literals, regexes are composed of sequences of characters, and, also like string literals, we need to escape the usual meaning of characters in regexes.

Once again, the backslash, is used as the escape sequence prefix.

THOMAS GUEST

Thomas is an enthusiastic and experienced programmer. He has developed software for everything from embedded devices to clustered servers. His website is http://www.wordaligned.org and you can contact him at thomas.guest@gmail.com

Ruby [4] embeds a powerful regex engine, so let's use Ruby for our regex examples. Here are some Ruby regex patterns:

```
/w/
/w+/
/\w+/
/"\w+"/
/\\/
```

Notice here that the *forward* slash, is used as a delimiter and is not part of the body of the regex pattern – just like the double-quote, `"`, was not part of the body our string literals.

What do these regex patterns mean?

- `/w/` matches the character 'w'.
- `/w+/` matches a sequence of one or more adjacent w's.
- `/\w+/` matches one or more adjacent 'word' characters.
- `/"\w*"/` matches a double-quote delimited sequence of one or more 'word' characters.
- `/\\/` matches a *single* backslash.

Did you notice that the backslash, gives the succeding `w` a special meaning? Did you notice that the `+` has a special meaning within a regex (it means one or more)? To match a literal `+`, we'd need to escape it like this: `/\+/` And did you notice that to match a literal backslash we must escape it? Good – but that was the easy bit!

Attempting to match a C++ string literal

Let's suppose we want to use our regex pattern matching on some C++ code. Now, matching a C++ string literal is going to be tricky. A first attempt, `/".*"/`, just won't do because the `.*` is *greedy* and will eat up everything until the final `"` in the text to be matched. So we might match too much:

```
char * s1 = "string", * s2 = "literal";
            ^---------match----------^
```

A non-greedy second attempt, `/".*?"/`, won't do either since it gets confused by an escaped double-quote in a string literal. So we might match too little:

```
char const * s =
  "The \"x\" in C++0x will probably be 9";
  ^match^
```

Correctly matching a C++ string literal

To properly match a C++ string literal we need to apply the following pattern: start with a double-quote; continue with a sequence of *either* characters which aren't the double-quote or the backslash *or* escape sequences; then finish with a double-quote.

Precisely what makes up a valid escape sequence is a little fiddly; there are octal and hexadecimal escapes, there are various whitespace characters, and there are unicode values. We can however compose a pattern using a suitable short-cut as follows:

```
/"([^"\\]|\\.)*"/
```

We can read this as: a string literal starts with a double quote, followed by any number of items which are:

- *either* not a double-quote or a backslash

- *or* are a backslash followed by any single character

and then finishes with a closing double-quote.

As you've probably spotted, we have to double up the backslashes in the regex pattern because the backslash is used as an escape sequence; i.e. a literal backslash is matched by the pattern `\\`.

Now let's do it in C++

I'll use the Boost [3] implementation since the compilers I have available don't support TR1 yet. We're going to need to construct a `boost::regex` using a pattern represented by a string literal. Which is where the problems start. Of course we can't write:

```
boost::regex const string_matcher(
   /"([^"\\]|\\.)*"/);
```

because we haven't passed a string literal to the `boost::regex` constructor. In order to pass a string literal we'll need to use double-quotes instead of forward-slashes and we'll have to escape the internal double-quotes. Let's try again:

```
boost::regex
  const string_matcher("\"([^\"\\]|\\.)*\"");
```

Oh dear – the error moves to run-time. We get an exception:

```
'Unmatched [ or [^'.
```

This is because the closing square bracket `]` has been escaped by the time it gets to the regex engine. Unfortunately the `\\`'s in the string literal contract to just single backslashes. We need to redouble them.

```
boost::regex const string_matcher(
  "\"([^\"\\\\]|\\\\.)*\"");
```

Here, each pair of backslashes has contracted to a single backslash by the time the regex engine sees it, which – believe it or not – is what's required.

This `string_matcher` works, but as code it is rather more cryptic than communicative.

A complete C++ string literal matcher

A complete program for you to try is shown in Listing 1.

Raw strings in Python

Unlike Ruby, Python [5] doesn't include support for regexes in the language itself. Instead, regex support is provided by the standard regular expression library [6].

Python's flexible string literals allow us to simplify the regex pattern, though. Here, we use a raw string [7], and we chose to delimit it with single-quotes so we don't need to escape the internal double-quotes.

```
string_literal_pattern = r'"([^"\\]|\\.)*"'
```

This is nice. Basically, raw strings leave the backslashes unprocessed. Raw strings aren't just restricted to regex patterns, though perhaps that's their most common use.

Raw strings in C++?

C++ doesn't support raw strings (at least, it doesn't support them yet, and I haven't found them mentioned in TR1) – but it does support wide-strings, indicated by the `L` prefix.

```
cpp_wide_string = L"this is a wide string";
```

Maybe if we switched the `L` for an `R` we could allow raw strings into C++? It would make regex patterns far more readable.

Verbatim strings in C++?

Alternatively ...

I've never used C# but googling suggests raw strings are supported and rather nicely named 'verbatim string literals'. C# uses the `@` prefix to indicate that a string literal is a verbatim string. Now, `@` isn't even part of the C++ source character set, so maybe this too would be possible.

Listing 1

```
#include <boost/regex.hpp>
#include <iostream>
#include <stdexcept>
#include <string>
int main(int argc, char * argv[])
{
  try
  {
    boost::regex const
     string_matcher("\"([^\"\\\\]|\\\\.)*\"");
    std::string line;
    while (std::getline(std::cin, line))
    {
      if (boost::regex_match(line,
                             string_matcher))
      {
        std::cout << line
         << " is a C++ string literal\n";
      }
    }
  }
  catch (std::exception & exc)
  {
    std::cerr << "An error occurred: "
     << exc.what();
  }
  catch (...)
  {
    std::cerr << "An error occurred\n";
  }
  return 0;
}
```

There's no escape

The proliferation of backslashes when we combine regexes and string literals is unfortunate. It could be worse. What if the backslash key had fallen off our keyboard? Remarkably – and, as far as I know, uniquely – C++ caters for this situation. A number source characters can be written as 'trigraphs' – sequences of three characters starting `??`. The backslash is one such character: it can be written as `??/`.

```
boost::regex const
    string_matcher("??/"([^??/"??/??/??/??/]|??/"
                   "??/??/??/.)*??/"");
```

For completeness, we could also lose the `|`, `[` and `]` keys.

```
boost::regex const
    string_matcher("??/"(??(^??/"??/??/??/??/"
                   "??)??!??/??/??/??/.)*??/"");
```

The string literal used to initialise `string_matcher` *is* valid, but the regex wouldn't match it properly. I'll leave the fix as an exercise for the reader. ■

References

[1] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf
[2] http://boost.org
[3] http://www.boost.org/libs/regex/doc/index.html
[4] http://www.ruby-lang.org "Ruby Website"
[5] http://python.org
[6] http://docs.python.org/lib/module-re.html
[7] http://docs.python.org/ref/strings.html

Loading a Container with a Range

Paul Grenyer looks for the simple solution.

Introduction

In this article I am going to look at the different methods of loading a C++ container with a closed range of numbers. A closed range of numbers includes the final number. Therefore the range 0 to 10 would include the number 10. The container I am going to use for the examples is vector. The container could equally be a list or another ordered container.

I recently came up against the problem of loading a vector with a range and expected it to be easy. The obvious method of loading the vector is:

```
std::vector< int > myVec;
for (int i = 0; i < 10; ++i )
{
  myVec.push_back( i );
}
```

However I wanted to write a function that would load a vector with both incrementing and decrementing ranges. Wrapped in a function, and with a **typedef** added for clarity, the example above is a good implementation for incrementing ranges:

```
typedef std::vector< int > VectorOfInt;
...

VectorOfInt LoadClosedRange(int first, int last)
{
  VectorOfInt result;
  for (int i = first; i != last; ++i)
  {
    result.push_back(i);
  }
  return result;
}

...

LoadClosedRange (0, 10);
```

However, as soon as it is used with a decrementing range, the loop over fills the container with everything but the expected range:

```
LoadClosedRange(10, 0);
```

So I am going to look at the different ways of implementing **LoadClosedRange** so that it works for both incrementing and decrementing ranges and look at some test code along the way.

Those of you with efficiency in mind will have noticed that returning a vector from **LoadClosedRange** is not the most efficient way to do things as it requires each element of the result to be copied on return. This is a valid argument, but one I am going to ignore for the relatively small ranges I am going to be dealing with.

PAUL GRENYER

Paul has been a member of ACCU since 2000. He founded ACCU Mentored Developers and has written a number of articles for CVu and Overload. Paul now contracts at an investment bank in Canary Wharf. He can be contacted at paul@paulgrenyer.co.uk

Listing 1

```
typedef std::vector< int > VectorOfInt;

namespace
{
  const int inc[] = {0,1,2,3,4,5,6,7,8,9,10};
  const int dec[] = {10,9,8,7,6,5,4,3,2,1,0};

  const VectorOfInt goingUp(inc,
    inc + sizeof(inc) /
    sizeof(inc[0]));

  const VectorOfInt goingDown( dec,
    dec + sizeof( dec ) /
    sizeof( dec[0] ) );
}
```

Testing

I am going to use the Aeryn [1] testing framework to unit test each of the **LoadClosedRange** implementations to make sure they handle both incrementing and decrementing ranges. I can also use Aeryn's context object facility to use the same test code for each **Load** implementation.

The easiest way to test if a **Load** implementation has loaded the right range into a vector is to compare it to another vector which has the correct range.

returning a vector from LoadClosedRange is not the most efficient way to do things

Therefore the first things I need to create are some vectors with an incrementing and decrementing range. This is easy to do at compile time, as the range is known then, using a couple of arrays, as shown in Listing 1.

The best way to compare two objects with Aeryn is to use the **IS_EQUAL** test condition macro. It uses **operator==** to compare objects. If two objects are not equal but are streamable, it streams them to the Aeryn report interface. Vector supports **operator==**, but isn't streamable. However, it is easy to add streamability to a **VectorOfInt** by implementing an **operator<<** at file scope (in the global namespace):

```
inline std::ostream& operator<<(
  std::ostream& out, const VectorOfInt& range)
{
  std::copy(range.begin(),
            range.end(),
            std::ostream_iterator< int >(out, " "));
  return out;
}
```

Aeryn uses a class contributed by Anthony Williams to detect streamable types. For user defined types it must be specialised. The specialisation for **VectorOfInt** looks like Listing 2.

Listing 2

```
namespace Aeryn
{
  namespace Williams
  {
    template<>
    struct IsStreamable< Range::VectorOfInt >
    {
      static FalseType testStreamable
        (const StreamableResult&);

      static TrueType testStreamable
        (...);

      static const bool result = true;
    };
  }
}
```

IS_EQUAL uses **IsStreamable::result** to determine if the types it is comparing are streamble. The above specialisation makes sure that it always determines that **VectorOfInt** is streamable. Separate test functions can be written to test incrementing and decrementing ranges (see Listing 3).

Listing 3

```
typedef VectorOfInt (*FuncPtr)(int,int);

void TestLoadInc(FuncPtr load)
{
  VectorOfInt vecToTest = load(0, 10);
  IS_EQUAL(goingUp, vecToTest);
}

void TestLoadDec(FuncPtr load)
{
  VectorOfInt vecToTest = load(10, 0);
  IS_EQUAL(goingDown, vecToTest);
}
```

A pointer to the **LoadClosedRange** function is passed to both functions and used to load a vector. **TestLoadInc** tests the range 0 to 10 using **goingUp**. **TestLoadDec** tests the range 10 to 0 using **goingDown**.

Now, instead of writing a separate test function for each **Load** implementation they can simply be specified as a context object when the test is registered (Listing 5).

For more details on registering tests and using context objects see the Aeryn User Guide [2].

Listing 4

```
VectorOfInt LoadClosedRange1(int first, int last)
{
  VectorOfInt result;
  for (int i = first; i != last; ++i)
  {
    result.push_back(i);
  }
  return result;
}
```

Listing 5

```
TestCase load1Tests[] =
{
  TestCase("LoadClosedRange1 (for loop) - TestLoadInc",
           FunctionPtr(TestLoadInc, &LoadClosedRange1)),
  TestCase("LoadClosedRange1 (for loop) - TestLoadDec",
           FunctionPtr(TestLoadDec, &LoadClosedRange1)),
  TestCase()
};
```

Listing 6

```
VectorOfInt LoadClosedRange1(int first, int last)
{
  VectorOfInt result;
  if (first < last)
  {
    for (int i = first; i != last + 1; ++i)
    {
      result.push_back(i);
    }
  }
  else
  {
    for (int i = first; i != last - 1; --i)
    {
      result.push_back(i);
    }
  }
  return result;
}
```

Load Implementation 1 – for loop

Now that the test code is in place it's time to revisit the **LoadClosedRange** function (Listing 4).

Plugging **LoadClosedRange1** into the test gives the following output.

```
Test Set : loadTests

- LoadClosedRange1: (for loop) - TestLoadInc
c:\...\test_load.cpp(20): '0 1 2 3 4 5 6 7 8 9 10 '
does not equal '0 1 2 3 4 5 6 7 8 9 '
-------------------------------------------------

Ran 1 test, 0 Passed, 1 Failed.
```

This is the classic off by one error. It is clear from the test vectors, **goingUp** and **goingDown**, that the range loaded should include both the **first** value and the **last** value. This can be achieved by adding 1 to **last**. This will fix the incrementing test, but will overwrite the vector when running the decrementing part of the test. The solution is to test for an incrementing or decrementing range and introduce a second **for** loop (see Listing 6).

The test will now pass:

```
Test Set : LoadClosedRange1 (for loop)

 - LoadClosedRange1 (for loop) - TestLoadInc
 - LoadClosedRange1 (for loop) - TestLoadDec
-------------------------------------------------

Ran 2 tests, 2 Passed, 0 Failed.
```

This is a good simple solution that gets the job done. However, it has two significant drawbacks:

1. Duplication of code. If the **for** loops need changing for any reason, for example the values are inserted into the container using a different function, then the code must be changed in two places.
2. If an **unsigned int** were used instead of an **int**, **last – 1** where **last** is initially zero, will cause **last** to overflow.

In order to remove the duplication the same **for** loop must be made to increment as well as decrement. This can be done by modifying the loop's expression:

```
const int direction = last < first ? -1 : 1;
for (int i = first; i != last; i+= direction)
{
  result.push_back(i);
}
```

This, of course, reintroduces the off by one error and cannot be solved by simply adding `1` to `last` as decrementing ranges would miss the final two numbers in the sequence. So when the sequence is incrementing `1` must be added to `last` and when the sequence is decrementing `1` must be taken away from `last`. This can be achieved as follows:

```
const int direction = last < first ? -1 : 1;

for (int i = first;
     i != last + direction;
     i+= direction)
{
  result.push_back(i);
}
```

This is a reasonable solution. However it doesn't solve the `unsigned int` drawback described above. This can be solved removing the adjustment of `last` and placing a call to `push_back` after the `for` loop:

```
const int direction = last < first ? -1 : 1;
int i = first;
for (; i != last; i+= direction )
{
  result.push_back(i);
}
result.push_back(i);
```

This of course reintroduces the duplicate code drawback as `push_back` is used in two places, but not as severely as before as there is only a single loop. This drawback can be reduced further by introducing a function (Listing 7).

Load implementation 2 – STL solution

The for loop presents a reasonable solution, but shouldn't there be something in the STL to make this easier and more expressive? There is certainly an STL algorithm that could be argued to make the solution more expressive: `std::generate_n`. The C++ standard [3] describes it as follows as shown in the sidebar.

The implementation of the `LoadClosedRange` function using `std::generate_n` requires a functor to maintain some state and looks like Listing 8.

The functor maintains and returns, by way of the function operator, the current number in the range and increments or decrements it through each iteration. The first and last values in the range are used to determine if the range is incrementing or decrementing.

Listing 7

```
inline void append(VectorOfInt& cont, int value)
{
  cont.push_back(value);
}

VectorOfInt LoadClosedRange1(int first, int last)
{
  VectorOfInt result;
  const int direction = last < first ? -1 : 1;

  int i = first;
  for (; i != last; i+= direction)
  {
    append(result, i);
  }
  append(result, i);
  return result;
}
```

Listing 8

```
class RangeGen
{
public:
  RangeGen(int first, int last)
   : current_(first), step_(first < last ? 1 : -1)
  {
  }

  int operator()()
  {
    int result = current_;
    current_ += step_;
    return result;
  }

private:
  int current_;
  int step_;
};

VectorOfInt LoadClosedRange2(int first, int last)
{
  VectorOfInt result;
  RangeGen rangeGen(first, last);
  std::generate_n(std::back_inserter(result),
                  std::abs(last - first) + 1,
                  rangeGen);
  return result;
}
```

C++ Standard Definition

25.2.6 Generate

```
template<class ForwardIterator, class Generator>
void generate(ForwardIterator first,
              ForwardIterator last,
              Generator gen);

template<class OutputIterator, class Size, class Generator>
void generate_n(OutputIterator first,
                Size n,
                Generator gen);
```

1. Effects: Invokes the function object `gen` and assigns the return value of `gen` though all `range [first, last)` or `[first, first + n)`.
2. Requires: `gen` takes no arguments, `Size` is convertible to an integral type (4.7, 12.3).
3. Complexity: Exactly `last - first (or n)` invocations of `gen` and assignments.

The first argument of `std::generate_n` updates the supplied container via an iterator. `std::back_inserter` is used here as new elements must be inserted into the vector.

The second argument is the number of times the functor's function operator is invoked and therefore the number of numbers in the range. Section 25.2.6/1 from the standard, in the sidebar, describes the half-open range behaviour of `std::generate_n`. We need to include all the numbers in the range, which is determined by the difference between `first` and `last`. 1 must be added to accommodate the half-open range behaviour of `std::generate_n`.. As `first` can be less than `last` the difference can be negative, so `std::abs` is used to ensure it is always positive.

From a functionality point of view this is the perfect solution to the problem. It handles incrementing and decrementing ranges, the off by one error and there is no code duplication. It separates the concerns of generating the range and inserting it into the container. The only draw back is the extra

code required for the functor. However that is often the way when using the STL.

Listing 9

```
namespace
{
  ...
  const int negativeInc[]
     = {-5,-4,-3,-2,-1,0,1,2,3,4,5};

  const int negativeDec[]
     = {5,4,3,2,1,0,-1,-2,-3,-4,-5};

  const VectorOfInt negativeGoingUp(
    negativeInc,
    negativeInc +
    sizeof( negativeInc ) /
    sizeof( negativeInc[0] ) );

  const VectorOfInt negativeGoingDown(
    negativeDec,
    negativeDec +
    sizeof( negativeDec ) /
    sizeof( negativeDec[0] ) );
}

void TestLoadNegInc( FuncPtr load )
{
  VectorOfInt vecToTest = load( -5, 5 );
  IS_EQUAL( negativeGoingUp, vecToTest );
}

void TestLoadNegDec( FuncPtr load )
{
  VectorOfInt vecToTest = load( 5, -5 );
  IS_EQUAL( negativeGoingDown, vecToTest );
}
```

Testing revisited

I have presented here two different ways of loading a vector with a range and tested each method using an incrementing and decrementing range. However, each range contains only positive numbers and the function signature for `load` can take positive or negative integers. Therefore some tests should be added to cater for ranges with negative numbers. This is easily done and is shown in Listing 9.

After adding and registering the new test functions all the tests still pass showing that the `Load` implementations can all handle ranges with negative numbers as well. For complete test coverage tests for corner cases such as, but not restricted too, `load(0,0)` and `load(0,1)` should also be added. This is left as an exercise for the reader.

Conclusion

Either of the two methods of loading a vector with a closed range presented here will do the job acceptably. The for loop is the simplest implementation and the main advantage is that it uses the least code. However I favour the STL solution as it is free to take advantage of any implementation defined optimisations as per Scott Meyers Effective STL [4] item. ■

References

[1] Aeryn – A C++ Testing Framework: http://www.aeryn.co.uk

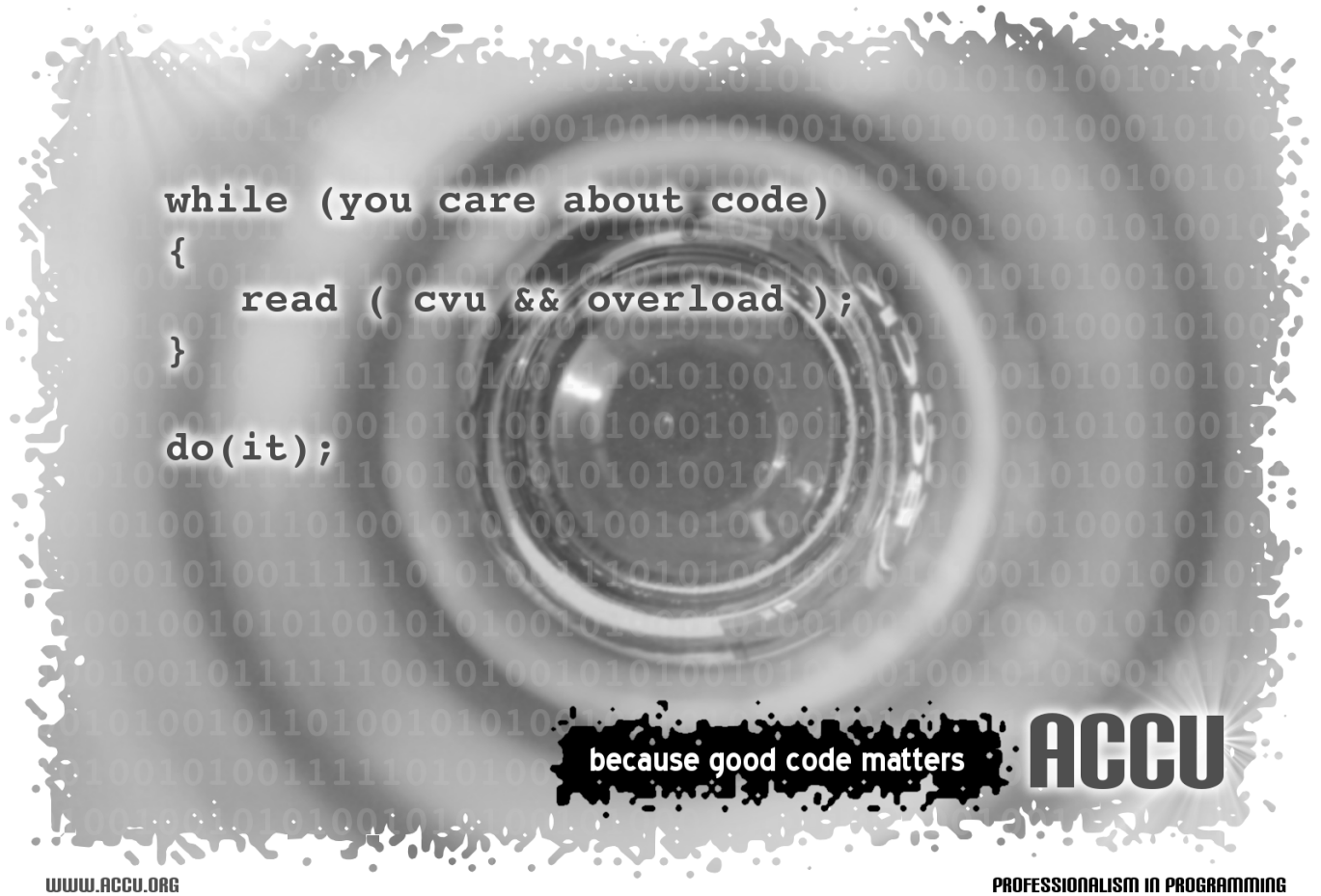
[2] *Aeryn User Guide*: http://aeryn.tigris.org/download/aeryn_user_guide.pdf

[3] *The C++ Standard*. ISBN: 0 470 845732

[4] *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library* by Scott Meyers. Item 43: 'Prefer algorithm calls to hand-written loops.' ISBN: 0-201-74962-9

Acknowledgments

Thank you to James Slaughter, Paul Smith, Frances Buontempo, Morten Boysen, Tim Penhey, Simon Sebright, Jez Higgins, Peter Hammond and Kevlin Henney for help with this article and background material.



WWW.ACCU.ORG

PROFESSIONALISM IN PROGRAMMING

Meet-Up Report

Ryan Alexander reports on the first London regional meeting.

As Jez mentions in "View from the Chair", ACCU as an organisation is getting more active in helping with local get togethers. The intention is to have a speaker talking on some hopefully relevant topic. This will just be another string to the ACCU bow.

A part of this formalisation process is to get reports back from those that attent the meetings, so other members can get an idea of what is going on, and hopefully either go to an existing one, or get motivated to start their own in their area. Here we have the first of the reports to filter back, form the first London meet-up.

Tim Penhey

The first ACCU London meeting!

On the first of November, ACCU had what will hopefully be the first of many London meetings. Hosting our first event were Richard Harris and Asti Osborne. Their flat in Bermondsey was a wondeful venue, spacious, and right on the river, a very enviable space indeed! I was quite surprised at how many chairs they were able to get in there.

I would say at a rough guess we had about 20 people in attendance including some faces familar to me including such as Paul Grenyer (who was also foolish enough to ask the new kid for a write up, how am I doing Paul?)

As I understand these talks have been quite some time in coming, since I'm a recent joiner myself I can't speak for anyone else as regards timing. For me, given that they started barely a month after I joined up, I thought the timing was quite considerate on their part, I don't have to feel like I missed anything.

Speaking of missing something, if you weren't there, you did. Starting us off in grand fashion for our first talk was none other than Kevlin Henney himself, presenting a slightly shortened, yet still quite rousing version of a talk titled A Critical View of C++ Practices. While I only recently joined, I did attend the ACCU conference in Oxford in 2005, and I believe I recall Kevlin speaking at that same conference on patterns. His style is well-paced, fast and dense, precisely the kind of thing you're looking for in conference that you've paid to go to. If you're going to hear him speak be prepared to keep up. It was of course, doubly challenging for me due to less experience with C++, however, I did still get a lot out of the talk, since the talk was on practices and there are lessons to be learned by any programmer in that area.

After burning our way through (some of) what is wrong with the state of C++, like any civilized group we retired to a nearby pub (also on the Thames, lovely view.) to continue our conversations in a (only slightly, did I mention how nice Richard and Asti's flat is?) more conducive environment. Sadly having been just on the tail end of a cold myself at the time, I needed to stick to less inebriating refreshments. I quite look forward to the next talk and hopefully being in better health to both enjoy the topics and proper conversations afterwards! ■

RYAN ALEXANDER

Ryan Alexander is a relatively new member of ACCU. He can be contacted at rnalexander@gmail.com

Advertise in C Vu & Overload

80% of readers make purchasing decisions, or recommend products for their organisations.

Reasonable rates. Flexible options. Discounts available to corporate members.

Contact ads@accu.org for info.

JOIN ACCU

You've read the magazine. Now join the association dedicated to improving your coding skills.

ACCU is a worldwide non-profit organisation run by programmers for programmers.

Join ACCU to receive our bi-monthly publications *C Vu* and *Overload*. You'll also get massive discounts at the ACCU developers' conference, access to mentored developers projects, discussion forums, and the chance to participate in the organisation.

What are you waiting for?

How to join
Go to www.accu.org and click on Join ACCU

Membership types
Basic personal membership
Full personal membership
Corporate membership
Student membership

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

Code Critique Competition # 43

Set and collated by Roger Orr.

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

Before we start

Remember that you can 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 also helps people living overseas who get the magazine much later than members in the UK and Europe.

A small change of name

You will see that the column as now simply "Code Critique Competition". We felt that the word 'Student' in the original title was discouraging some people from entering the competition (and perhaps from supplying possible code for critique).

Code Critique 42 entries

There is a string class that sometimes gives unexpected failures. Here's a simple test program that asserts with one compiler – but only in debug – and works with another. Please explain the assert failure and critique the code.

```
#include "MyStr.h"
#include <assert.h>

int main()
{
   MyStr s;
   assert( s == "" );
   s = "10";
   assert( s == "10" );
}
```

The header file for **MyStr** is:

```
#include <string.h>

class MyStr
{
public:
  MyStr(char const * str = 0);
  MyStr(const MyStr& str);
  const MyStr& operator=(const MyStr& str);
  ~MyStr() { delete [] myData; }

 bool operator==(const MyStr& d) const;
  friend bool operator==(const MyStr& lhs,
    char const * rhs);
  friend bool operator==(char const * lhs,
    const MyStr& rhs) { return rhs == lhs; }

 void alloc(unsigned N) const
  {
    MyStr *nc = const_cast<MyStr*>(this);
    char * od = nc->myData;
    nc->myData = new char[N];
    nc->mySize = N;
    copyStr(myData, od);
    delete [] od;
  }

 void copyStr(char* d, char const* s) const
  {
    if(s == NULL)
       d = NULL;
    else
       strcpy(d,s);
  }

  operator char *() const { return myData; }

  // more methods unused in test not shown
private:
  char * myData;
  unsigned  mySize;
};

inline MyStr::MyStr(char const * str)
: myData(NULL), mySize(0)
{
  if(str == NULL) return;
  unsigned newlen = strlen(str) + 1;
  alloc(newlen);
  copyStr(myData, str);
}

inline MyStr::MyStr(const MyStr& str)
: myData(NULL), mySize(0)
{
  if(str.myData == NULL) return;
  unsigned newlen = strlen(str.myData) + 1;
  alloc(newlen);
  copyStr(myData, str.myData);
}

inline const MyStr& MyStr::operator=(
  const MyStr& str)
{
  if(str.myData == NULL)
  { myData = NULL; mySize = 0; }
  else
  {
    unsigned newlen = strlen(str.myData) + 1;
    if(newlen > mySize)
      alloc(newlen);
    copyStr(myData, str.myData);
  }
  return *this;
}
```


ROGER ORR

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

```
inline bool MyStr::operator==(const MyStr& d)
  const
{
  if(myData == d.myData) return true;
  if(myData == NULL || d.myData == NULL)
    return false;
  return !strcmp(myData, d.myData);
}

inline bool operator==(const MyStr& lhs,
 char const * rhs)
{
  if(lhs.myData == NULL && rhs == "")
    return true;
  else if(lhs.myData == NULL) return false;
  else return !strcmp(lhs.myData, rhs);
}
```

From Nevin :-] Liber < nevin@eviloverlord.com>

Before diving into the code, let's take a step back and talk about strings.

In C, a string is modeled by a char*, which is either a pointer to characters terminated with a '\0', or a NULL pointer.

In the C++ Standard Library, a **`std::string`** is an object that keeps track of the characters as well as a count of the characters, and that count does not include a terminating '\0' (and '\0' characters may be embedded in the string).

Now, here are the class invariants for **`MyStr`**:

- **`MyStr::myData`** is a pointer to the character buffer, which is allocated on the heap.
- **`MyStr::mySize`** is a count of characters, including a terminating '\0'.
- Special case: when **`MyStr::myData == NULL`**, **`MyStr::mySize == 0`** (and vice versa).
- Combining the last two bullet items, the empty string is represented by **`MyStr::mySize <= 1`**.

This is a fairly complicated model. My first inclination is to simplify this (either use a terminating '\0' or a count of characters but not both, only one representation of an empty string, etc.). However, the problem states that there are more public methods that are not shown here, so I can't really change it without breaking those methods.

On to code issues

General comment: there is one more invariant which is assumed by the code as written: namely, there are no embedded '\0' within the string, other than the terminating one. I'll note places where this assumption is made. Given the methods we have seen so far, the only legal way this could happen is someone modifying the buffer they got by calling **`MyStr::operator char*() const`** (to address this and other issues, a little bit later in this article I will propose changing the signature of this operator). It is also reasonable to assume that some of the unspecified public methods might also do this.

`void MyStr::alloc(unsigned N) const`: This is a routine for growing the allocated space in a string. It should not be a const member function, because it modifies all the member variables, using a very roundabout way (const_casting this to the non-const pointer nc).

`void MyStr::copyStr(char* d, char const* s) const`: since this function is not dependent on any member variables, it really should be declared static, not **`const`**. It also has a bug, in that the statement **`d = NULL;`** effectively does nothing; it should read **`*d = NULL;`**. It also assumes that **`d != NULL`** and that the buffer which **`d`** points to is large enough to hold a copy of **`s`**.

`MyStr::operator char*() const`: Since the contents of the buffer are part of the class invariants (either the pointer is NULL or the buffer terminates with a '\0'), it is not a good idea to let the caller modify the buffer. This should really be declared **`MyStr::operator char const*() const`**.

`MyStr::MyStr(MyStr const& str)`: Assumes no embedded '\0' characters.

`unsigned MyStr::mySize`: Since **`mySize`** is really filled in by **`strlen()`**, its type should be **`size_t`**, not unsigned.

`MyStr const& MyStr::operator=(MyStr const& str)`: there are lots of issues here.

The return type of **`operator=`** should be **`MyStr&`**, not **`MyStr const&`**. This could prevent **`MyStr`** objects from being stored in standard containers (see Exceptional C++, item #41, for more details).

If **`myData != NULL && str.myData == NULL`**, there is a memory leak, as the memory pointed to by **`myData`** is not **`delete []`**-ed.

If **`newLen < mySize`**, **`mySize`** is not set to the correct value. Assumes no embedded '\0' characters.

`bool MyStr::operator==(MyStr const& d) const`: Assumes no embedded '\0' characters.

`bool operator==(MyStr const& lhs, char const* rhs)`: NULL dereference when **`rhs == NULL && lhs.myData != NULL`**.

`rhs == ""` compares two pointers, not the buffers to which they are pointing.

Those pesky asserts

An assert only fires in debug because assert is a macro, not a function. In a release build, the code inside the assert is never executed (and hence one must be very careful not to put code with side effects inside an assert).

So why does the first assert in main sometimes fire, depending on the compiler? It is a combination of the **`rhs == ""`** in **`bool operator==(MyStr const& lhs, char const* rhs) const`**, and that some compilers pool string literals. What pooling string literals does is whenever two string literals are exactly the same, they are only stored once, and will end up having the same address for everyone who uses them.

Here is the call chain

```
MyStr s;
  s.myData = NULL;
  s.mySize = 0;

assert( s == "" );
  operator==(s, "");
  if (lhs.myData == NULL && rhs == "") return true;
      if (true && "" == "") return true;
```

If strings are pooled, **`"" == ""`** will return true because both instances of the literal **`""`** have the same address. This causes **`operator==`** to return **`true`**, and the assert doesn't fire.

If strings aren't pooled, **`"" == ""`** returns **`false`**, and we move on to the next line of code:

```
  else if (lhs.myData == NULL) return false;
      if (true) return false;
```

`operator== returns false`, and the assert fires.

Fixing the code

There are lots of subtle issues with this code. I also found it difficult to reason what was going on in a lot of it. Because of this, I am choosing to rewrite it instead of just fixing the bugs. Hopefully, I've made it a bit simpler (no casts, simple helper functions with good names, etc.) However, I am still constrained to the class invariants of the original.

Here is my rewrite, with comments explaining the reasoning behind my changes:

```
#ifndef MYSTR_H_
#define MYSTR_H_
#include <algorithm>    // for std::swap
#include <cassert>      // for assert
#include <string.h>

class MyStr
{
  // Switched the order of the member
  // variables, because I am going to use the
  // value of mySize in the construction of
  // myData
  // Changed the type of mySize from unsigned
  // to size_t, to match strlen(...)
  size_t mySize;
  char* myData;

  // Helper function which determines if
  // string is empty based on its size
  static bool empty(size_t theSize)
  { return theSize <= 1; }

  // Helper function to compare a MyStr with a
  // char const* and its length
  // If the strings are both empty, return
  // true; otherwise,
  // If the strings have a different size,
  // return false; otherwise,
  // Compare the buffers via memcmp
  bool equalTo(char const* yourData, size_t
    yourSize) const
  {
    return empty(mySize) && empty(yourSize) ||
      mySize == yourSize &&
      !memcmp(myData, yourData, yourSize);
  }

  // Helper function to determine the size of
  // a char const* which is consistent with
  // the class invariants of MyStr
  // if !data, size == 0; otherwise,
  // size == strlen(data) + 1
  static size_t size(char const* data)
  { return data ? strlen(data) + 1 : 0; }

  // Helper function to new [] a buffer and
  // copy the srcData into it
  // Assumes srcSize <= destSize
  // Only allocates the buffer when
  // 0 < destSize; otherwise returns NULL
  // When srcSize < destSize, the rest of the
  // buffer is uninitialized
  static char* newCopy(char const* srcData,
    size_t srcSize, size_t destSize)
  {
    char* destData(destSize ? new
      char[destSize] : 0);
    memcpy(destData, srcData, srcSize);
    return destData;
  }

  // Helper function to new [] a buffer and
  // copy the srcData into it,
  // when we want the new buffer to be the
  // same size as srcSizse
  static char* newCopy(char const* srcData,
    size_t srcSize)
  { return newCopy(srcData, srcSize, srcSize); }

public:
  // All the work has been moved into the
  // member initializer lists
  // In general I find this far less error
  // prone
  MyStr(char const* myStr = 0)
  : mySize(size(myStr))
  , myData(newCopy(myStr, mySize))
  {}

  // All the work has been moved into the
  // member initializer lists
  // In general I find this far less error
  // prone
  MyStr(const MyStr& that)
  : mySize(that.mySize)
  , myData(newCopy(that.myData, mySize))
  {}

  // Public helper function which swaps the
  // guts of a MyStr
  // This is non-throwing
  void swap(MyStr& that)
  { std::swap(mySize, that.mySize);
    std::swap(myData, that.myData); }

  // Returns a MyStr&, not a MyStr const&,
  // so that it can be used in standard
  // library containers
  // Strong exception safety guarantee
  // Easy to reason what is happening here, as
  // all the work is now in the
  // constructor and destructor
  // 1. Copy construct the parameter into rhs
  // 2. Swap our guts with the guts of rhs
  // 3. When rhs goes out of scope, our old
  // guts are cleaned up
  // In general, I find this far less error-
  // prone
  MyStr& operator=(MyStr rhs)
  { swap(rhs); return *this; }
    ~MyStr()
  { delete [] myData; }

  // Make all the operator== friends, so that
  // they have the same form
  // Calls the helper equalTo to do the work
  friend bool operator==(MyStr const& lhs,
    MyStr const& rhs)
  { return lhs.equalTo(rhs.myData,
    rhs.mySize); }
  friend bool operator==(MyStr const& lhs,
    char const* rhs)
  { return lhs.equalTo(rhs, size(rhs)); }
  friend bool operator==(char const* lhs,
    MyStr const& rhs)
  { return rhs == lhs; }

  // Made this return a char const* instead of
  // a char*
  // Note:  this change may break other code;
  // breakage compiler detectable
  operator char const*() const
  { return myData; }

  // Made this a non-const member function
  // Note:  this change may break other code;
  // breakage compiler detectable
  // Asserts if mySize > N
  // Strong exception safety guarantee
  // Uses the same type of swapping logic as
  // operator=
  void alloc(unsigned N)
```

```
  {
    assert(mySize <= N);
    if (mySize < N)
    {
      MyStr that;
      that.myData = newCopy(myData, mySize,
        that.mySize = N);
      swap(that);
    }
  }

  // I consider this an unsafe interface
  // I am only leaving it in for backwards
  // compatibility (since it is a public
  // function)
  // Asserts when d == NULL
  // Made this a static member function
  static void copyStr(char* d, char const* s)
  { assert(d); if (s) strcpy(d, s);
    else *d = '\0'; }
};

namespace std
{
  // Specialization of std::swap to use the
  // more efficient, non-throwing
  // implementation
  // Useful in things like standard library
  // containers and algorithms
  template<> inline void
  swap<MyStr>(MyStr& lhs, MyStr& rhs)
  { lhs.swap(rhs); }
}
#endif  /* MYSTR_H_ */
```

I'm just hoping the above code won't be the subject of the next Student Code Critique... :-)

From Simon Sebright <simon.sebright@ubs.com>

Me: Ah, hello again

Student: [Looks sheepish] Hello

[Bit of a pause]

Student: Erm, I was... did you get my email about my problem?

Me: I did indeed

Student: Do you think you might be able to help me?

Me: No, not at the moment, but I will help you to help yourself. What have you done so far to diagnose the situation?

Student: Er, well, I compiled it fine

Me: Fine, huh? With what settings:

Student: Settings? Well, default I suppose

Me: Right, go and find out about the warning levels the compiler has. Set it to maximum, tell the compiler to treat warnings as errors and then let's see how far we get

Student: Max level, warnings as errors, OK. Yes, alright then, I'll, um, give it a go and get back to you

Me: Well, hopefully you won't need to!

Student: Well I did what you said and…

Me: Don't do it 'cause I said, do it to give yourself the best chance of the compiler helping you. So, what did you find?

Student: Well, I got 4 warnings. Three about converting `size_t` to `unsigned int`, but I can ignore those, and one...

Me: Hang on a minute, why do you think you can ignore those warnings?

Student: Well, I looked at the code and worked out that the ranges are compatible

Me: OK, suppose you are in your first job and this code is being run in company's product. What if you leave, and someone else takes over, they'll just have to do the same analysis again.

Student: I'll put a comment in then

Me: NO!!! Say it in code. Change the compiled code to get rid of it. Right, what about that 4th warning?

Student: Oh, yes, the other one. Something about comparing an address and a string constant. Not sure what it means

Me: Right, this is the line `if(lhs.myData == NULL && rhs == "")`, is it not? What's it doing?

Student: Well, it's just checking to see if we've got nothing ourselves, and we are being asked to compare with an empty string

Me: No, I said what is it doing, not what do you think it is doing. Look at that bit `rhs==""`. What does that do?

Student: Well, it sees if `rhs` is empty?

Me: What is `rhs`?

Student: It's a string

Me: Really?!

Student: [beginning to twig that something is up] it's a char const star

Me: Right! And what are you comparing it to?

Student: A string. No hang on, a const char… ah, it's comparing the addresses of the strings! How the hell did it ever work at all?

Me: Well, when you put a string literal in your code, the compiler will put it in the compiled code, and can choose where it goes, as long as the content is corrrect. This applies even for this empty string, because the memory needed for this is not in fact zero chars but…?

Student: Er, ah, one, for the null terminating character

Me: Exactly, so when you write `""` in your code the compiler has a little bit of memory with one null character in it, but there is no guarantee that all places you write `""` will have the *same* place in memory, and we said earlier we comparing memory here.

Student: Ah, so I need to check the content of the string instead of where it is. Doh, that's what I'm doing further down the function. Great, got it now, thanks, bye

Me: Woh, not so fast!

Student: Huh?

Me: You do realise that when you are featuring anything to do with ACCU, that you will get full advice on everything you are doing wrong, don't you?

Student: ACCU, what's that?

Me: Find out and join, it'll do you good. Anyway, here is a list of things I noted when looking at your project. If I don't note these, I'll never win…

1. warning C4130: '==' : logical operation on address of string constant && rhs == "" (we've dealt with this now)
2. assignment operator: `myData = NULL` – memory leak if already assigned
3. strCopy:

   ```
   if(s == NULL)
   d = NULL;
   ```

   useless assignment, string is unaffected, local variable changed. Do you mean `*d = '\0'` to have an empty string?
4. one of the global `==` operators doesn't need to be a friend
5. duplicate logic in constructors and assigment operator, length calculation, `strcmp`. Could have implemented with common helper. Read up on implementing assignment with `copy` constructor and `swap()`
6. self assignment not protected (see comment about `swap()`, this then wouldn't be a problem)
7. implementation functions public - `alloc()`, `copyStr()`: why?

8. `alloc()` separated from `strCopy()`, these two should always be done together, enforce it
9. use of `int` variable by default, be explicit
10. if you work in a professional company, they won't appreciate classes called `MyXxxx`. Give it a useful, neutral name
11. have a think about how your member `operator==` will behave if one side has `myData` of `NULL`, and the other has non-null `myData` which is empty.
12. you could have implemented the comparison operators with one common function (after null checks), consider how you would have gone about making comparison case-insensitive, for example. In fact, if you had made the member `operator==` a non-member, then either side could be implicitly converted from `const char*` to `const MyStr&`, so you wouldn't have needed the other two!

Student: Er, right, it's not that straightforward writing a string class, is it?

Me: No, even the stl string comes in for a slating, although more because of its huge interface than the fact that it doesn't work!

Student: Hmmm! [Looks a little sheepish]

Me: Still, it's a good exercise to hone your C++ knowledge, as we've seen, but in real life, prefer to use an implementation which is provided!

Student: Yes, thanks. I'll go away and study this list. Bye…

Me: Bye

From Michal <michal_hr@yahoo.pl>

I see two errors in this code:

1. In the inline `bool operator==(const MyStr& lhs, char const *rhs)` most interesting is the first "if" statement: `if (lhs.myData==NULL && rhs=="")`...

   The second comparison `(rhs=="")` is incorrect.

   Why ? The following example explain this:

```
const char* s1="something"; //1
const char* s2="something"; //2
if (s1==s2) {
  cout<<"Equal"<<endl;
} else {
  cout<<"Not equal"<<endl;
};
```

   The output from the code above is implementation dependent. If compiler allocated single `something` string then s1 and s2 point to the same memory address and output is `Equal`.

   But it's also legal for compiler to allocate two copies of `something` string and then s1 and s2 point to two different memory addresses. Because `operator==` is called at line `assert(s=="");` the result depends on how the compiler allocates `""` string from the above line and `""` string from the `rhs==""` statement.

   If there is only one `""` string allocated then `operator==` returns `true` and assert will not fail. Otherwise `false` is returned and program is stopped at line `assert(s=="")`. I believe that in the debug mode compiler makes two copies of `""` string and therefore `operator==` fails. To avoid this problem there should be: `if (lhs.myDate == NULL && !strcmp(rhs,""))`...There is not problem with second assert as it's `lhs.myData` is not `NULL` so string are compared with `strcmp` function.

2. The second problem is with `operator char *() const {return myData;}` Const function should not allow to change object state. This function doesn't meet this criteria as it returns `char *` pointer which may be modified. This way we can modify const object.

   To avoid it this operator should return `const char *: operator const char *() const {return myData;}`

   Anyway it doesn't have any impact on program output as this operator is not called.

I have also a few minor comments:

- header file code should be surrounded with directives:

```
#ifndef INCLUDED_MYSTR
#define INCLUDED_MYSTR
/* header code */
#endif
```

  This way we don't have trouble (multiple definitions) if header is included multiple times.

- assignment operator should return reference to the value which is not reference to const. Additionally it should check if there is self assignement:

```
inline MyStr& MyStr::operator=(const MyStr&
   str) {
if (this == &str) return *this;
  ....
}
```

- operator new should be either non-thrown:

```
nc->myData = new (std::nothrow) char[N];
 if (nc->myData == NULL)  ....
//handle lack of memory
```

  or should catch `std::bad_alloc` exception:

```
try {
  nc->myData = new char[N];
  ...
} catch (const std::bad_alloc &ref) {
  //handle lack of memory
}
```

Commentary

I came across this code in production use and was rather taken with it; it was amazing the code worked as well as it did.

My own preference would be to delete the class completely and replace it with `std::string` but sadly this sort of refactoring was not possible for various, mostly non-technical, reasons.

My role was to port the existing code base from one compiler to another and it was while doing this that I came across the item which I turned into the assert failure.

As Nevin points out, the behaviour of the `operator==` is highly dependent on whether the compiler does string pooling. The code I was porting broke at runtime because the source compiler shared string literals for `""` across the whole program whereas the compiler I was porting to only shared string literals within the same object file.

Fixing code like this is notoriously difficult. What makes it so hard?

Firstly the original code is hard to understand and so working out what it actually does is non-trivial.

Secondly string classes are fundamental and so any changes to the implementation involves recompiling and retesting the whole code base. Also, as Simon said, "it's not that straightforward writing a string class, is it?".

Thirdly in some places (such as in comparing strings against `""`) the code relies on the 'broken' behaviour, so attempts to change things might cause other repercussions.

As an example, changing the `operator char*() const` to `operator const char*() const` broke some code elsewhere as the overload set considered was different, but the problem only surfaced at runtime.

I made some use of `private:` in the migration to a slightly safer implementation – making unsafe operators private and then fixing all the code that failed to compile to use new, safer, methods. One of the difficulties with operator overloading is how hard it is to search through the code for all uses of the overloaded operator!

The Winner of CC 42

The winner of CC 42 is: Simon Sebright, by a short head. One of the things I liked about his critique was the inclusion of helps for the student to learn the technique of finding and resolving bugs for themselves.

Code Critique 43

(Submissions to scc@accu.org by 8th January)

This code is designed to provide a simple encryption of plain text, and **main** is a test harness than encrypts and then decrypts the text.

These seems to be a problem with displaying the encrypted text for strings with spaces:

```
crypt "a test"
```

Please comment on the specific bug and the code as a whole.

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

void strreverse( char *src, char *dest) {
  int i=0, j;
  for( j= strlen(src)-1; j>=0;j-- )
  {
    dest[i++] = src[j];
  }
}

bool encrypt(const char *input, char *output,
             bool encode)
{
  char buffer[80] = {0};

  const char *src = input;
  char *dest = buffer;

  srand( strlen( src ) );
  int seed = rand();

  while (*src)
  {
    int minVal,maxVal;

    minVal = maxVal = 0;
    int charVal = (int)*src;

    if( charVal >= 33 && charVal < 48 )
    {
      minVal = 33; maxVal = 48;
    }
    else if( charVal >= 48 && charVal < 58 )
    {
      minVal = 48; maxVal = 58;
    }
    else if( charVal >= 58 && charVal < 65 )
    {
      minVal = 58;  maxVal = 65;
    }
    else if( charVal >= 65 && charVal < 91 )
    {
      minVal = 65; maxVal = 91;
    }
    else if( charVal >= 91 && charVal < 97 )
    {
      minVal = 91; maxVal = 97;
    }
    else if( charVal >= 97 && charVal < 123 )
    {
      minVal = 97; maxVal = 123;
    }
    else if( charVal >= 123 && charVal < 127 )
    {
      minVal = 123; maxVal = 127;
    }
    else if (charVal < 33 || charVal > 126 )
    {
      return false;
    }

    int range = maxVal - minVal;
    int key = maxVal % range;
    int delta = range - key;
    if ( encode )
    {
      if( charVal < maxVal - key )
        sprintf( dest, "%c", (charVal + key) );
      else
        sprintf( dest, "%c",
                 (charVal - delta) );
    }
    else
    {
      if( charVal >= minVal + key )
        sprintf( dest, "%c", (charVal - key) );
      else
        sprintf( dest, "%c", (charVal + delta) );
    }
    *dest++;
    *src++;
  }
  *dest = '\0';

  char revBuffer[80]={0};
  strreverse(buffer,revBuffer);
  strcpy(output,revBuffer);
  return true;
}

int main( int argc, char **argv )
{
   for ( int i = 1; i < argc; i++ )
   {
      char *input = argv[i];
      char output[80];
      encrypt( input, output, true );
      printf( "%s\n", output );
      encrypt( output, input, false );
      printf( "%s\n", input );
   }
}
```

Prizes provided by Blackwells Bookshops & Addison-Wesley

Standards Report

Lois Goldthwaite announces the forthcoming publication of a revised standard.

Nothing focuses the mind better than a deadline! The lead story from Portland is that the international C++ committee have made a firm commitment to publish a revised standard by fall 2009.

Counting backwards from the publication date gives the milestones that have to be met along the way. The ISO process requires three ballots by national standards bodies of the world before a document becomes a standard, and each of those ballots requires several months to complete – – partly a hangover from the days when documents and official letters were shipped around the world by boat, and partly to give national experts the time to devote some serious study to their decision. That is why producing standards in ISO takes longer than in groups whose formal process can only be described as dash-it-off-and-ship-it-while-the-ink-is-still-dripping-we'll-fix-glaring-problems-when-we-issue-the-next-revision-in-six-months.

The first step for C++09 – let's quit calling it C++0x now – is what is called a Registration ballot. This is accompanied by a Committee Draft which shows the complete subject matter to be covered by the final standard. This vote is to see whether the voting nations believe the subject coverage is suitable for the topic. The document need not contain the complete text, or even a first draft of the complete text, but if any major topics are added after this, or if any promised topic is cut, the final standard may fail to win approval. This document was approved at the Portland meeting; more on this below.

The second ballot is on a document called the Final Committee Draft. This contains near-final text, which may then be revised based on comments which accompany the votes. This stage of C++98 received so much feedback, leading to so many changes, that a second FCD ballot took place. The plan is to issue this document following next April's meeting in Oxford.

The last ballot is on a Final Draft International Standard. This is the final text, and if this gains approval only the smallest changes are allowed afterwards. If the Final Draft is complete by the April 2009 meeting (and gains approval by the national bodies, of course), there is a decent chance that publication could take place in the the fall of 2009.

With the approval of the registration document (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2135.pdf), we now have a pretty good idea of what C++09 will include. This document includes a number of "placeholder" sections, announcing that such-and-such a feature will be included, and pointing readers to committee papers discussing the topic. Here is one example:

> 14.9 Concepts
>
> This section is a placeholder. The next C++ standard is intended to include support for concepts. This feature is intended to provide language support for describing features of types, for example to express the container requirements tables in the C++ Standard Library as code that can be checked by the compiler. For more information and snapshots of current draft proposals still under discussion and development, see: [the following list of papers].

Of all the topics that have been discussed in recent years, what's in and what's out? Apart from concepts, the other major features are aimed at preparing C++ for a world in which concurrent multiprocessing is the normal environment (already there are mobile phones with dual-core processors, and laptop computers with them are common). These include a new memory model/abstract virtual machine specification for C++, a set of atomic types and operations to prevent undefined behaviour in concurrent programs without the overhead of ubiquitous defensive locking, support for local storage tied to a unique thread, and an API for launching and synchronising threads so that most programmers will never have to grapple with guru-level issues addressed by the earlier features in this sentence.

the other major features are aimed at preparing C++ for a world in which concurrent multiprocessing is the normal environment

Probably the biggest 'sex-appeal' factor comes from the decision to include garbage collection in C++09. Whether to use it is at the option of the programmer, since in some application areas (embedded programming, in particular), complete control over memory management is essential.

A big disappointment, to the UK panel at least, was the vote to exclude 'modules' from the document. This is a proposal from David Vandevoorde to bring a feature something like .Net assemblies to C++, with the aim of speeding up compilation (because header files would no longer be needed) and making libraries play nicer together (because internal names would be not just private but invisible, preventing unintentional symbol clashes). It must be admitted that this proposal was not so mature as the others, since there is no working implementation yet. However, WG21 simultaneously applied for permission to launch a project to produce a technical report on modules, which we hope will be published soon after the standard, and the UK panel will be supporting this project enthusiastically.

There are several other new features which you can expect to see in C++09, but which were judged too small to warrant special mention in the registration document. A near certainty is greater support for Unicode, and lambda functions are also likely to make it in.

The C committee also met in Portland the week after WG21, and in a surprise development have decided to start work on a revised version of the C standard. I hope to have more details for the next issue of C Vu.

The standards committees are meeting in Hawaii in October, with a full agenda, and in Oxford and London next April, immediately following the ACCU conference.

If you are interested in joining a UK panel, please write to standards@accu.org for more details. Attendance at meetings is not compulsory; much of the work is accomplished via email discussion. ■

LOIS GOLDTHWAITE

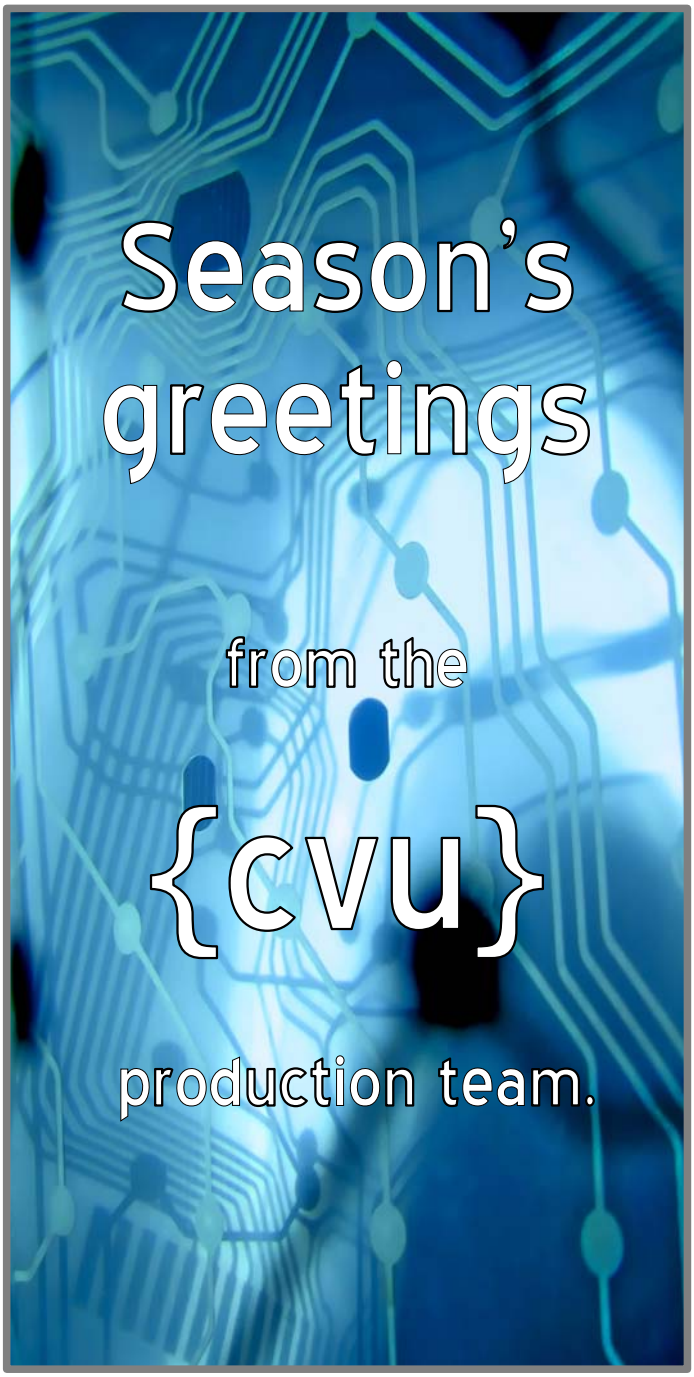
Lois has been a professional programmer for over 20 years. She is convenor of the C++ and Posix standards panels at BSI. One of her hobbies is representing the UK at international standards meetings!
Lois can be contacted at standards@accu.org.uk

Obfuscated Code Competition

One of the sections that I used to really enjoy in the old C++ Report was the obfuscated C++ section at the end of the magazine. Perhaps it appealed to my perverse side. The following C code (posted to accu-general some while ago) prints out the words to the traditional English song 'The 12 Days of Christmas'. I think that everyone will agree that this code is somewhat obfuscated.

The challenge is to write the same program in a different language but keep the obfuscation. It could be obfuscated in an entirely different way – perhaps with template metaprogramming, perhaps just writing it in perl. The most entertaining and perverse will make an appearance in the next issue.

Entries to scc@accu.org before 8th January 2007.




```
main(t,_,a )
char
*
a;
{
  return!

0<t?
t<3?

main(-79,-13,a+
main(-87,1-_,
main(-86, 0, a+1 )


+a)):

1,
t<_?
main( t+1, _, a )
:3,

main ( -94, -27+t, a )
&&t == 2 ?_
<13 ?

main ( 2, _+1, "%s %d %d\n" )

:9:16:
t<0?
t<-72?
main( _, t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,"
"/w{%+,/w#q#n+,/#{l,+,/n{n+,/+#n+,/#;#q#"
"n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';"
"dq#'l q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{n"
"l]'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw' i"
";# ){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K"
" {rw' iK{;[{nl]'/w#q#n'wk nw' iwk{KK{nl"
"]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlw"
"b!/*de}'c ;;{nl'-{}rw]'/+,}##'*}#nc,',#"
"nw]'/+kd'+e}+;#'rdq#w! nr'/ ') }+}{rl#'"
"{n' ')# }'+}##(!!/")
:
t<-50?
_==*a ?
putchar(31[a]):

main(-65,_,a+1)
:
main((*a == '/') + t, _, a + 1 )
:

0<t?

main ( 2, 2 , "%s")
:*a=='/'||

main(0,

main(-61,*a, "!ek;dc i@bK'(q)"
"-[w]*%n+r3#l,{}:\nuwloca-O;m"
" .vpbks,fxntdCeghiry")

,a+1);}
```

Bookcase

The latest roundup of book reviews.

It seems that the dropping of the book prices has been met with just about complete acceptance, so I will be carrying on with that from now.

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.

You will notice some of the books listed on the website are getting a tad long in the tooth. These are pretty much pointless to review and so the following will happen. After the dates below, books will be dumped.

Those in 2004 will be disposed of on May 1st. Those in 2005 will be disposed of on November 1st.

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.

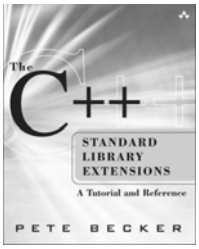
I must thank Blackwells and Computer Bookshop for their continued support in providing us with books.

C++

The C++ Standard Library Extensions: A Tutorial and Reference

by Peter Becker, ISBN: 0321412990, published by Addison Wesley Professional

Reviewed by Thomas Guest



The standard C++ library is a fine thing but there are some notable omissions and weaknesses: there are no hashed containers, few smart pointers, no standard regular expression library; support for gluing functions and algorithms could be improved on; and so on. The first C++ Library Technical Report (TR1) addresses these issues and many more. In 2006 the TR1 library was approved by ISO, and you can already find TR1 implementations. Pete Becker's book provides a comprehensive and accurate reference guide for the TR1 library.

The book styles itself as the perfect companion to Josuttis *The C++ Standard Library*, and that's what I hoped for – a book that would cut through the standardese and provide clear instructions on how I could benefit from TR1. In the main, it succeeds. There's plenty of example code, and there needs to be – TR1 gives the standard C++ library a sizeable boost. The code is clearly written and described, and available for download from the author's website. The examples I tried (using GCC 4.01) worked, though I had to fiddle a little with include paths. Pete Becker has first-hand knowledge of implementing TR1, giving this book an authorative tone. I can imagine this book becoming *the* TR1 Book and I would certainly recommend it.

I do have some niggles, though. More attention could have been given to the layout. The code examples often break awkwardly across pages and some form of syntax highlighting would have made them more readable. I can't understand why the output from these programs was either omitted or buried in a paragraph of explanatory text. Code comments were abused throughout the book to provide a running commentary. E.g.

```
tuple<> t0;
// default constructor

tuple<int> t2(3);
// element initialised to 3
```

I realise this is common practice in programming books, but I'd like to see authors and publishers find a better way to annotate code.

These are niggles, though. My only real complaint was that many of the examples failed to show the benefits of using TR1. Much of TR1 is designed to make C++ easier to use; it's easier to manage dynamically allocated objects, it's easier to bind function arguments, it's easier to wrap functions for use in standard algorithms. The examples showed how to get TR1 code compiling and linking, but sometimes failed to explain why.

Reviewed by: Pete Goodliffe

Rating: Highly recommended

The 'tr1' extension to the standard C++ library is a large and comprehensive addition with much useful functionality. It draws heavily from the Boost libraries (www.boost.org) and should become an important part of any C++ programmer's arsenal. At the time of writing, few compilers currently provide a version of 'tr1' in their C++ offerings, but almost all the tr1 functionality is available from www.boost.org (albeit in the 'boost' namespace, not 'tr1') so all of the material in this book is immediately useful.

Enough of the background. This book is a comprehensive guide to a large library. It comes from a respected and authoritative author. It is a genuinely good tutorial and excellent reference.

The book reads well and introduces material at a well-judged pace, even if the author does have an unfortunate predilection for footnotes. (If you've read Becker's CUJ/DDJ columns you'll understand this! At least in book form these footnotes have a minor blessing – they're at the bottom of the page, so you don't have to keep turning pages to read them.)

The questions at end of each chapter are interesting. They are a great way for reader to gauge how much they understood. However, I wonder how many people will actually read them. Practicing programmers need the reference information available immediately, and will dwell less on these. Often the first question in each chapter asks you to decipher the evil template error messages that typically result from a single type error in the kind of monstrous template constructions that tr1 allows you to construct. Telling. This book is rather dense for a university level course text, but C++ programming courses may find it useful.

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

- **Computer Manuals** (0121 706 6000)
  www.computer-manuals.co.uk
- **Holborn Books Ltd** (020 7831 0022)
  www.holbornbooks.co.uk
- **Blackwell's Bookshop**, Oxford (01865 792792)
  blackwells.extra@blackwell.co.uk

The book is marred by a few unfortunate typos and errors that weren't picked up at editorial stage. In such a good book this is a real shame. The tutorial could also do with a little more practical application; the questions have a little of this, but good examples of some library usage in real code would be immensely beneficial.

I have yet to see a better book on this subject, and the bar has definitely been raised for further offerings. Highly recommended for all C++ programmers.

Beyond the C++ Standard Library

by Bjorn Karlsson, publisher: Addison Wesley, 388 pages, ISBN: 0-321-13354-4

Reviewed by Pete Goodliffe

Rating: Recommended

The Boost libraries have become increasing important for C++ programmers, especially as many of their libraries have been adopted into the 'tr1' standard extension and are slated for inclusion in the next revision of the C++ standard.

Karlsson's book acts as as a tutorial and reference to some of the basic Boost libraries. These days Boost is far too large to cover comprehensively in a single volume (indeed, some of the larger Boost libraries have entire books to themselves). So the author picks the most immediately useful and applicable parts of Boost (you can't really call them the "most important" libraries, though). This is clearly a somewhat arbitrary choice, but it has been made well.

The book covers Boost's smart_ptr, conversion, operators, regex, any, variant, tuple, bind, lamda, function and signals libraries. On the whole the discussion is clear and well thought out. The regex coverage is not as clear or deep as in Pete Becker's *The C++ Standard Library Extensions*.

Every so often Karlsson mixes in some interesting side notes or describes an advanced C++ idiom within a chapter's description of a Boost library. It is a shame that more was not made of this. These parts could have been formatted differently, put in sidebars, or separated visually somehow to make them stand out. Overall I find the presentation of the book a little odd and hard to read (and sometimes inconsistently applied).

In a book released in 2006, I'd like to see more mention of what's in tr1. It is mentioned briefly in the preface and noted in passing in library descriptions (and not at all in the index).

It's inevitable that many of the libraries Karlsson has chosen have been incorporated into C++'s tr1 (and pretty soon into an official revision of the standard itself). This book will date as the preferred versions become the standard library implementations, rather than the deprecated Boost versions.

It also seems slightly strange that mention of Boost is relegated to the book's subtitle rather than in its main title. Nevertheless, this is a useful and important book, and a great introduction to the magic of Boost.

Recommended.

Java

Java After Hours - 20 projects you'll never do at work

by Steven Holzner (0-672-32747-3), Sams

Reviewed by Paul Thomas

Most programming books these days seem have titles that bear little relation to their content. Publishers cynically sprinkle words like 'Advanced' or 'OO' around because it gets the books on company shelves. *Java After Hours*, however, does exactly what it says on the tin. This is not code to impress your boss or your colleagues. It's not even 'macho' code to impress hackers, but it is probably at about the right level for 'play code' if you want to dip your toes into different areas of Java technology.

The layout and style of the 10 chapters is easy to follow. Code is incrementally introduced with with lots of explanation and a little less padding that usual. The programs are built up in roughly the way you would create them so you aren't just reading through a source file from beginning to end. As key class are encountered, mini references are provided to give an overview of important methods. The documentation for these methods is often laughable (e.g. `getName()` - Gets the name) but it still works.

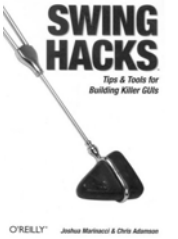
The projects gradually build up a body of knowledge in a few core areas of Java technology. The blurb focuses on what the applications do, so you might be interested to know that by the end of the book you'll have more than a passing familiarity with AWT (including Graphics2D, the awt.image package and the awt.Robot), Swing, SWT, JSP and Sockets. Threads are covered, but I have serious reservations about the material. Synchronisations are not covered, instead we are told that 'The standard technique for communicating with threads is with Boolean flags'. Bizarrely, the author even confesses in a sidebar that he once crippled a client's servers with runaway threads. Read this as 'ignore everything I say about threads'.

It's a real shame, because I would like to recommend this book to people who need to branch out from the Java they use at work, but the technical inaccuracies are quite serious. I have already heard people say they based real-world code on the techniques in this book. As well as the problem with threads, the author seems to be claiming that HTTP Basic Authentication is more secure than HTML forms with password controls. It all makes me very uneasy about what else is flawed.

Not Recommended.

Swing Hacks

by Joshua Marinacci & Chris Adamson (0-596-00907-0), O'Reilly



Reviewed by Paul Thomas

Despite myself, I really like this book. The name might not inspire confidence in project managers, but it's deceptive. Like the cookbook format, it consists of 100 sections devoted to recipes or 'hacks'. Many are simply explanations of how to use the Swing framework in the way it was intended but some are truly evil hacks that use reflection techniques to delve into internal object structure. All of them help the reader gain a deep understanding of the Swing framework and advanced Java techniques.

This isn't the kind of book you can simply read end-to-end (I did try). The titles are just clear enough that you might be able to find information when you are stuck with a problem. Really though, this is one to keep nearby for whenever you have the spare time. I freely confess that I haven't read it all, but I'm dipping into it every now and then and it still has masses to offer.

If there's one problem with the book, it's that I am now tempted to waste project time adding translucent drag-and-drop where it isn't needed. But if you have enough self control, you'll find something that will make life easier on just about every project. If you work with Swing, you NEED this book.

Highly Recommended.

Miscellaneous

Home Networking – A visual do-it-yourself guide

by Brian Underdahl, published by Cisco Press, ISBN 1-58720-127-5

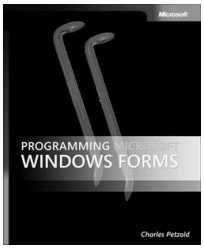
Reviewed by Ian Bruntlett.

In this book's favour, it is a slim volume with plenty of practical examples and helpful photographs, split up into three manageable parts (Introduction, Starting your network, Enhancing your network). It discusses wired networks, wireless networks and combinations of the two. It is a partisan work, however – always recommending a "Linksys model X" or a "Linksys model Y" etc. throughout the book. In terms of doing the reader favours, though, it falls flat on its face. True, this book is the best I've seen so far in terms of introducing a reader to the topic but this book lacks both a glossary and a

further reading section. I feel that Cisco Press should commission a follow up book that digs a bit deeper and discusses the protocols involved.

Verdict: Excellent introduction, technological dead end.

Programming Microsoft Windows Forms (A streamlined aproach using C#) 2005 Edition



by Charles Petzold, published by Microsoft Press, ISBN 0-7356-2153-5,

Reviewed by Steve Love

Recommended (with some reservations)

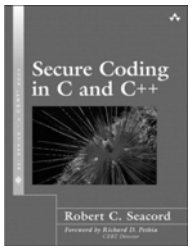
First up, if you're familiar with other titles by Charles Petzold, then the basic content of this book will come as no surprise – lots of code listings and an explanation of what's going on in the code. All the code is written 'Petzold Style' (his term) which means no generated code from the Microsoft Form Designer. That at least makes the examples *much* more concise. Nevertheless, a couple of the examples cover 7 pages or more.

A lot of the book is given over to general Windows Forms programming, but it does cover some aspects new to Windows Forms in Visual Studio 2005 – notably dynamic layout and tool strips. Also covered in reasonable detail are custom controls and data bound controls. This last topic especially interested me, because it's not just about binding controls to a database; you can bind controls to each other.

My reservations come from two observations. The first is that he makes almost constant reference to his previous publication *Programming Windows with C#*, which makes me feel this book is incomplete without that companion. The second is that despite depending on references to that other book, the whole first 2 chapters of this one probably cover much of the same background information – around one quarter of the book.

That apart there is little to criticise on the technical content. One or two hideous hacks when the code strays a little off the main road, and a few seemingly out-of-place tutorials (how to create a test program for a DLL project, using ClickOnce deployment). This being Charles Petzold, even in C# the code uses a kind of Hungarian Notation (my favourite is szImage – no not a null-terminated string, but a Size struct!), and the code is pretty much written in a straight line, so to speak, but this isn't a book about style, it's about Windows Forms techniques. I wouldn't describe this book as a 'must have' for Windows Forms developers, because a lot of it rehashes old ground, but I expect most Windows developers would find it useful.

Secure Coding in C and C++



by Robert C. Seacord, published by Addison Wesley, 341 pages, ISBN: 0-321-33572-4

Reviewed by Pete Goodliffe

Rating: Neutral

There are fewer books devoted to writing secure code than many other topics in the software engineering world. And this is clearly a field that many more developers need education in.It sits in Addison Wesley's CERT/SEI series, so it appears authoritative. Indeed, many chapters are written in collaboration with the author's CERT colleagues. But does it live up to this and is it a worthwhile addition to the (albeit small) security party?

As the title suggests, the book focusses on security in C and C++ alone. It opens with a dry but customary survey of the appalling state of software security in modern code and goes on to lambast C and C++ (with some merit, the majority of insecure code is written in C and C++ - but does that make them insecure languages?). Seacord draws primarily on Windows and Linux as examples, which is a good choice, and each chapter shows practical examples of exploits in real code, which helps ground the discussion.

The book is split into sections covering main areas of code vulnerability, including string-based attacks, pointer subterfuge, memory management and so on. They are each reasonable discussions of the topic, but to fully understand the material the reader must understand C/C++ reasonably proficiently beforehand. Student programmers would probably need a more gentle paced introduction to security.

My main reservation about this book is its C++ coverage. There are really no good examples of secure C++ code – despite the book's title. Seacord mostly deals with C-related problems (which are indeed many and numerous). The book does not satisfactorily describe how the C++ idiom is inherently safer than C (and too often we are talking about the mythical 'C/C++' language, as if the two languages are one and the same).

Good programming in C is necessarily very different from good programming in C++. This is not made clear at all, and few places show some of the more secure idioms available to the C++ programmer. For example, the entire description of C++ iostream output is two paragraphs and a small quote from Meyer's *Effective C++*. This is in a 40-odd page chapter on I/O security (mainly discussing printf/scanf vulnerabilities and buffer overflow problems). There is also an interesting claim that it is 'as easy' to create a safe abstract type in C as in C++. Hmm. In a modern security book that claims to be about C++ this doesn't seem justifiable.

This is not a bad book, but perhaps does not live up to the expectations I had of it.

Human-Computer Interaction

by Serengul Smith-Atakan, published by Thomson, 204 pages

This book is part of the Thomson FastTrack series, and appears to be a one-semester university course wrapped up in a book. It is accompanied by web-supported course work and exam preparation, and downloadable power point slides it says. This turns out to be pretty much one big pdf file with review questions and answers. There is one particular on-line case study, that of a ticket machine at a station which appears to be hijacked wherever possible. I read the original article by a group of academics and came away thinking, "all very well, but are we really paying these people to publish this stuff". It really felt like someone trying to make their publication list look bigger, without really having anything new to say.

It is split into 12 chapters, 10 if you discount the introduction and review chapters. The remaining 10 discuss natural computing, user modelling, user-centred system design, some very basic psycology, evaluation techniques, current and future trends of technology and universal access. I say discuss, it was really a matter of defining these things very briefly so that a student could be asked, "what is natural computing?", etc.

I was disappointed with this book. I was also worried about it. If students are using this as reference material and coming out of university thinking they know their stuff, then it doesn't bode well. True, there are reading lists with each chapter, exercises and review questions, but I felt the general level of information was very poor. It was a very repetetive experience. I kept thinking I was reading the same paragraph, because there were so many similar sentences, really making it feel like someone had a lecture course and coerced it into a book simply by padding it out with waffle. For example, chapter 5 is called Task Analysis. OK so far. 5.1 Introduction. Ok. 5.2 Task Analysis Hmm 5.3 What is Task Analysis Hmm!! 5.4 Approaches to Task Analysis, and each of these sections had numerous sentences begining Task Analysis...

More important, some of the content was either very out-dated or just plain wrong. One concept mentioned is User-Centred System Design, or USCD. This is presented as a methodology for producing software which is going to be usable. Their justification for it is to compare it to the waterfall model, which they do twice!

It's basically an iterative process, involving users (XP anyone?), BUT, in each increment, they are developing a 'prototype'. At the end of the iterations, when the users are happy, the 'prototype' is 'installed'. Really, leave software development to people who know about it. I have no problem with iterations or prototyping, but they need to be used appropriately.

ACCU Information
Membership news and committee reports

accu

View From The Chair

Jez Higgins
chair@accu.org

Do you want some free money? Conditions apply [*].

One of the many things I've learned since taking on the Chair is that I need to keep better meeting notes. Ahead of the last committee meeting on September, I'd been reviewing the notes of the previous meeting and discovered an action on me to mail accu-general and announce in C Vu the free money the committee agreed to give away.

Those present at the AGM will no doubt remember Reg Charney's impromptu speech on receiving his honorary membership. The committee bestowed honorary membership on Reg, in large part, for the sterling work he had done in setting up and running a monthly meeting in Silicon Valley, which attracts first class speakers. Reg spoke eloquently, not about what he'd put in, but what he'd gained from the meetings. I know many people were inspired by what he said and are thinking about starting meetings in their own area. Indeed, what I hope is the first of many meetings was held just a few days ago.

At our meeting following the AGM, the committee agreed to make funds available to help with some of the costs that might be incurred setting up such meetings. That might mean the cost of, as Reg suggested, leafleting your local technical bookshop, printing posters, equipment costs, initial room hire, things of that nature.

We haven't set a budget, nor drawn up a list of what's in or out. What might be needed is going to depend very much on circumstances. Some people might envisage something along the Silicon Valley model, with a programme of speakers at a regular venue. Others might want something rather more informal. We are, therefore, going to consider each application on its own merits.

Note also that we can't fund meetings on an ongoing basis. This is intended as seed capital, to get things going.

If you think you would like some free money, email me at chair@accu.org.

[*] That condition? You have to write about your meeting for CVu.

Membership Report

David Hodge and Mick Brooks
membership@accu.org

Our membership now stands at 864. For the first time in the last few years we are showing a net gain of members, currently 18. Since the beginning of the year we have gained 170 new members. Mick and I have developed a spec to allow the membership system to be run from the website. You will be able to change your details online and the membership secretary will do less of the things that I did on a regular basis as they will be automated from the website. The quote for this work should be approved in November and should all be up and running by the AGM in April, Mick Brooks will be taking over as membership secretary in April. He has already started part of the job as the new members will now be getting the welcome letter and their first set of journals from Mick. Although there will be a lot of the membership operations on the website there will still be a human being to contact if you have any queries.

Website report

Allan Kelly

I hope members will excuse any lack of activity on the website in the least few months. Personally I've not been paying much attention lately, my mind has been elsewhere – what with getting married and getting caught up in company downsizing. Fortunately others have continued to push ahead.

David's membership report makes two interesting points. Firstly, association membership is rising. While we can't prove that this is due to the new website many of us hoped a new website would provide a boost to membership. The challenge now is to keep both the content and appearance of the new site fresh and active.

Secondly, new work is focusing on the association membership system. Hopefully by integrating this with the website we can simplify the membership secretaries work and provide a better (faster) service to our members. While there is other work to (e.g. migrate the mailing lists, integrate ACCU USA) the membership system should (hopefully) be the last big piece of work.

By the time you read this C Vu should be available online to members only. We have not plans to make C Vu available to non-members.

We have yet to start advertising on the website. As I mentioned last month, ACCU needs to find an advertising officer who can co-ordinate online and offline sales.

Book Reviews (continued)

She goes on to describe the different ways of prototyping, Throwaway, Evolutionary and Incremental. That latter is the funniest; we are told that the system is built as separate components, with the final product being released as a series of components, each release adding a component until the complete product is delivered. What utter rubbish!

That said, I did come away with one or two new things. One is Jakob Nielson's set of 10 heuristics to consider when designing a system. Also, distinguish between 'useful', 'usable' and 'accessible', the latter term hinting at another important point, that of considering not just one, or the 'typical' user, but consider your design for all users. A lot of emphasis is also given to catering for disabled users.

But, I could have got that from a few-page discussion. The only reason I would suggest anyone buys this book is if they happen to be on that university course, and then I would emphasise to go off and find some in-depth sources of information. The website has it at £19.99. On second thoughts, I would recommend that you borrow your friend's copy.

Regular Expressions Pocket Reference. (2003)

by Tony Stubblebine, ISBN 0-596-00415-X pp93

Reviewed by Mark Easterbrook.

Conclusion: Recommended (but not for beginners).

This pocket reference begins with an overview of Regular Expressions and continues with a look at each of the main languages and tools that support REs, including many examples. It would be a useful reference for an expert, but of most value at intermediate level. This is only a pocket reference so a beginner would be better off with a more comprehensive book such as *Mastering Regular Expressions* by Jeffrey Friedl.