The magazine of the ACCU

www.accu.org

{cvu}

Volume 21 Issue 4 September 2009 £3

Features

Don't Ignore That Error!
Pete Goodliffe

Java Dependency Management with Ivy
Paul Grenyer

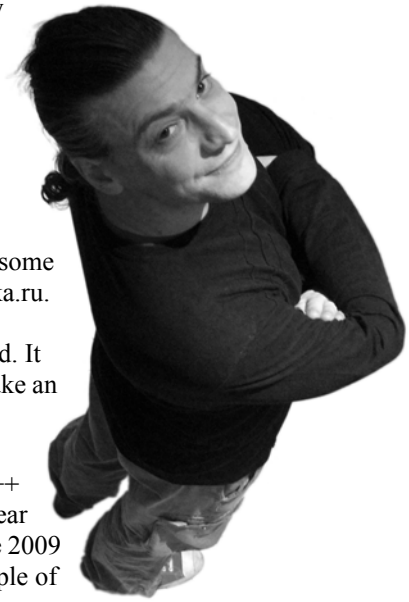
Interpreting Custom Unix Shell Scripts in C
Ian Bruntlett

Hunting the Snark
Alan Lenton

Bjarne Stroustrup
What is C++0x?

{cvu}
Volume 21 Issue 4
September 2009
ISSN 1354-3164
www.accu.org

ACCU Archaeology



The call for participation at the ACCU Conference 2010 came in to my inbox a short while ago. It was followed fairly closely by the announcement of the one-day conference at Bletchley Park, inspired by the legendary Enigma code-breakers.

I first joined ACCU around the time of the last ACCU Autumn Conference – actually called the Java and C/C++ (JaCC) Conference, held at the Oxford Union in September 1999. I didn't attend, but a bit of hunting brought me the session details. The old Russian ACCU mirror site still has some references, which you can find at http://www.accu.informika.ru. The schedule, as far as programming languages went, was predominantly C++, with Java and C fighting a close second. It would be a couple of years until C# and Python began to make an appearance. In 2001, JaCC became the ACCU Spring Conference, which is now just the ACCU Conference.

That the ACCU has diversified from its origins in C and C++ is a great thing, driven by its members: what they want to hear about at the conferences, and read about in the journals. The 2009 Autumn Conference at Bletchley Park is an excellent example of that, having been driven and organised entirely by members Astrid Byro and Alan Lenton. The world of computer software development (which is, after all, what most if not all of us are about) is a rich and varied thing. No one of us can know about all of it, but collectively, as an organisation, the members of the ACCU represent a huge amount of knowledge in one area or another.

If there's something you'd like to see in the journals (CVu *or* Overload), then let us know. Even better, write it, and share it with everyone. Many articles are written by experts, but yet more are written by non-experts who want to share their experiences and pass on what they've learned.

And if there's something you want to hear about at the conference, then submit a proposal; there's no better way of learning about a thing than giving a presentation on it.

This is *your* magazine, the conference is *your* conference, and the ACCU is *your* organisation. It is, in the end, what you make of it. 'By Programmers, For Programmers'..

STEVE LOVE
FEATURES EDITOR

The official magazine of ACCU

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

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

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

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

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

DIALOGUE

16 Desert Island Books
Paul Grenyer introduces Jon Jagger.

18 Code Critique #59
The winner of last week's competition.

26 Insirational (p)articles
Frances Buontempo introduces Linda Rising's inspiration.

26 ACCU Security – Yesterday, Today and Tomorrow
ACCU announce a 1-day conference at Bletchley Park.

REGULARS

27 Bookcase
The latest roundup of ACCU book reviews.

28 ACCU Members Zone
Reports and membership news.

FEATURES

3 Don't Ignore That Error!
Pete Goodlliffe never fails to manage failure.

4 Java Dependency Management with Ivy
Paul Grenyer looks into Ivy to help with Java dependencies.

7 Hunting the Snark (Part 4)
Alan Lenton investigates engineering in software.

8 What is C++0x?
Bjarne Stroustrup examines some of the changes to the C++ language for C++0x.

14 Interpreting Custom Unix Shell Scripts in C
Ian Bruntlett learns how to write an interpreter for a custom script language.

15 A Game of Dice
Baron Muncharris sets a challenge.

SUBMISSION DATES

C Vu 21.5: 1st October 2009
C Vu 21.6: 1st December 2009

Overload 94: 1st November 2009
Overload 95: 1st January 2010

WRITE FOR C VU

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

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

ADVERTISE WITH US

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

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

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

COPYRIGHTS AND TRADE MARKS

Don't Ignore That Error!

Pete Goodliffe never fails to manage failure.

Settle yourself down for an apocryphal bedtime story. A programmer's parable, if you will...

I was walking down the street one evening to meet some friends in a bar. We hadn't shared a beer in some time and I was looking forward to seeing them again. In my haste, I wasn't looking where I was going. I tripped over the edge of a curb and ended up flat on my face. Well, it serves me right for not paying attention, I guess.

It hurt my leg, but I was in a hurry to meet my friends. So I pulled myself up and carried on. As I walked further the pain was getting worse. Although I'd initially dismissed it as shock, I rapidly realised there was something wrong.

But, I hurried on to the bar regardless. I was in agony by the time I arrived. I didn't have a great night out, because I was terribly distracted. In the morning I went to the doctor and found out I'd fractured my shinbone. Had I stopped when I felt the pain, I'd've prevented a lot of extra damage that I caused by walking on it. Probably the worst morning after of my life.

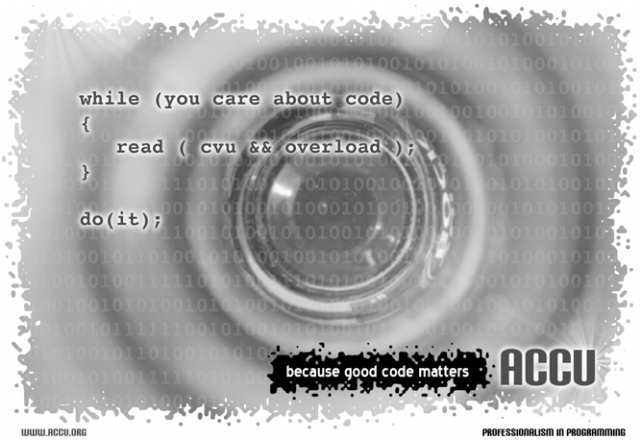
Too many programmers write code like my disastrous night out.

Error, what error? It won't be serious. Honestly. I can ignore it. This is *not* a winning strategy for solid code. In fact, it's just plain laziness. (The bad sort.) No matter how unlikely you think an error is in your code, you should always check for it, and always handle it. Every time. If you don't, you're not saving time, you're storing up potential problems for the future.

The mechanism

We report errors in our code in a number of ways, including:

- **Return codes**. A function returns a value. Some of which mean 'it didn't work'. Error return codes are far too easy to ignore. You won't see anything in the code to highlight the problem. Indeed, it's become standard practice to ignore some standard C functions' return values. How often do you check the return value from `printf`?
- `errno` This is a curious C language aberration, a separate global variable set to signal error. It's easy to ignore, hard to use, and leads to all sorts of nasty problems – for example, what happens when you have multiple threads calling the same function?



- **Exceptions** are a more structured language-supported way of signalling and handling errors. And you can't possibly ignore them. Or can you? I've seen lots of code like this:

```
try {
  /// ...do something...
}
catch (...) {} // ignore errors
```

The saving grace of this awful construct is that it highlights the fact you're doing something morally dubious.

The madness

Not handling errors leads to:

- **Brittle code**, full of hard-to-find crashes.
- **Insecure code**. Crackers often exploit poor error handling to break into software systems.
- **Bad structure**. If there are errors from your code that are tedious to deal with continually, you have probably have a bad interface. Express it better, so the errors are not so onerous.

Why don't we check for errors?

Just as you should check all potential errors in your code, you must expose all potentially erroneous conditions in your interfaces. Do not hide them, and pretend that your services will always work.

Programmers must be made aware of programmatic errors. Users must be made aware of usage errors.

It's not good enough to *log* an error (somewhere), and hope that a diligent operator will notice an error and do something about it one day. Who knows about the log? Who checks the log? Who is likely to do anything about it? If program termination is not an option, ensure that problems are flagged up in an unobtrusive, but obvious and non-ignorable manner.

The mitigation

Why don't we check for errors? There are a number of common excuses. Which of these do you agree with? How you would you counter each of them?

- Error handling clutters up the flow of the code, making it harder to read, and harder to spot the 'normal' flow of execution.
- It's extra work and I have a deadline looming.
- I know that this function call will never return an error (`printf` always works, `malloc` always returns new memory, and if it fails we have bigger problems...)
- It's only a toy program, and needn't be written to a production-worthy level.

Conclusion

This is a very short article. It could be much, much longer. But doing so would be an error. The message is simple: Do. Not. Ignore. Errors.

Ever. ■

PETE GOODLIFFE

Pete Goodliffe is a programmer who never stays at the same place in the software food chain. He has a passion for curry and doesn't wear shoes. Pete can be contacted at pete@goodliffe.net

Java Dependency Management with Ivy

Paul Grenyer looks into Ivy to help with Java dependencies.

Along with the rich enterprise libraries that come as part of the language, one of the biggest advantages of Java is the vast number of third party libraries available. For example if you are writing an enterprise web application, GWT [1], Spring [2] and Hibernate [3] provide a framework with a huge amount of pre-existing functionality.

The size and number of dependencies grows as your application grows. GWT and Spring alone, without their dependencies, are more than 7MB. The ideal place to put dependencies is in a source control repository as part of your project so that when you or your continuous integration server check out the project for the first time all the dependencies are there. Then you don't have to go and get them and store them locally in a location that is agreed by the entire development team.

Storing the dependencies for a single project in a source control repository isn't too bad, provided there is plenty of room on the source control server.. However, if you have more than one project using the same or similar sets of dependencies the amount of space taken up in the source control repository starts to get a bit ridiculous. And then when a new version of a library comes out and you upgrade, even more space is wasted as the differences between binary jars cannot be detected, so the entire jar must be replaced.

Solutions to dependency storage

There are a number of ways to solve the problem of dependencies taking up too much space in a source control repository.

You could check your dependencies into a single place in the repository, which does mean they're still taking up some space, and then use a link (e.g. svn externals [4]) to each project so that when the project is checked out the dependencies are checked out as well. Although this would work well it can be a pain to set-up and maintain, especially if you link to individual jars.

Maven [5] allows you to specify which dependencies you need and automatically downloads them for you when you check your project out. This means you do not have to check your dependencies into your source control system, just maintain a configuration file. The drawbacks are that you're restricted to the third party libraries available via Maven and you have to buy into the whole Maven project layout and configuration.

Ivy [6] provides the dependency management features of Maven, without the need to buy into any particular project layout. The default locations to download dependencies from are the Maven repositories, but you can also specify other locations and set-up your own own local repositories. Ivy creates a local cache and retrieves dependencies from it every time a project is checked out or another dependency is added.

Ivy represents the best solution as you don't need to store any dependencies in your source control system and you're not restricted to a particular project layout. In this article I am going describe how to use Ivy to manage dependencies for a Java project using Ant [7] and the IvyDE [8] plugin for Eclipse [9].

PAUL GRENYER

An active ACCU member since 2000, Paul is the founder of the Mentored Developers. Having worked in industries as diverse as direct mail, mobile phones and finance, Paul now works for a small company in Norwich writing Java. He can be contacted at paul.grenyer@gmail.com

Listing 1

```
<path id="ivy.lib" >
  <pathelement location =
     "thirdparty\ivy\ivy-2.1.0-rc2.jar"/>
  </path>

<taskdef uri="antlib:fr.jayasoft.ivy.ant"
   resource="fr/jayasoft/ivy/ant/antlib.xml"
   classpathref="ivy.lib"/
```

Listing 2

```
import org.apache.commons.lang.WordUtils;

public class IvyAnt
{
  public static void main(String[] args)
  {
    final String msg = "hello, world!";
    System.out.println(
       WordUtils.capitalizeFully(msg));
  }
}
```

Using Ivy with Ant

To use Ivy with Ant, download the latest release of Ivy from the Ivy website, extract the Ivy jar file (at time of writing: `ivy-2.1.0-rc2.jar`) and either:

- copy it to Ant's `lib` directory (`ANT_HOME\lib`)
- copy it to another directory and use an Ant **`taskdef`** task to reference it (Listing 1)

The easiest way to demonstrate using Ivy with Ant is with a simple application that has a single dependency (Listing 2).

This application uses the **`apache.commons.lang`** library to correctly title-case a simple message. The basic Ant script (`build.xml`) for an application like this is as in Listing 3.

It only does two things:

1. Creates a `bin` folder for the compiled class files.
2. Attempts to compile java files into class files.

Listing 3

```
<project name = "IvyAnt"  basedir="."
   default = "compile" >
  <property name = "src.dir"
     value="${basedir}/src" />
  <property name = "bin.dir"
     value="${basedir}/bin" />

  <target name = "init">
    <mkdir dir = "${bin.dir}"/>
  </target>

  <target name = "compile" depends= "init">
    <javac srcdir="${src.dir}"
       destdir="${bin.dir}" fork="true">
    </javac>
  </target>
</project>
```

Listing 4

```
<?xml version="1.0" encoding="UTF-8"?>
<ivy-module version="2.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-
   instance" xsi:noNamespaceSchemaLocation=
   "http://ant.apache.org/ivy/schemas/ivy.xsd">

  <info organisation="purpletube.net"
     module="IvyAnt" status="integration"/>
  <dependencies>
    <dependency org="commons-lang"
       name="commons-lang" rev="2.0"/>
  </dependencies>
</ivy-module>
```

If run from the command line it fails as it cannot find the jar file for **apache.commons.lang**. Enter Ivy!

To get Ivy to download and cache dependencies for us, we need to tell it where to get them from. This is the purpose of the `ivy.xml` file (Listing 4).

The Ivy Quick Start Guide [10] describes the format of the ivy.xml very well, but in summary the **info** tag is used to give information about the module for which we are defining dependencies. In the **dependencies** section, the **org** and **name** attributes define the organization and module name of the dependency. The **rev** attribute is used to specify the revision of the module you depend on. Dependencies can be looked up in the Maven repository [11]. Once found, the Maven POM (Project Object Model – an XML representation of a Maven project held in a file), for example:

```
<dependency>
   <groupId>commons-lang</groupId>
   <artifactId>commons-lang</artifactId>
   <version>2.0</version>
</dependency>
```

can be converted to an Ivy dependency. Use the **groupId** as **organization**, the **artifactId** as **name**, and the **version** as **rev**.

Put the `ivy.xml` file in the same directory as your Ant `build.xml`. To get Ant to use Ivy and resolve dependencies the Ivy namespace and the Ivy retrieve task need to be added (Listing 5).

Running `build.xml` again will download and cache **apache.commons.lang** and its dependencies, but the build will still fail. Ignore the warning about a missing Ivy settings file for the time being. The settings file is only used if you want to use settings other than the default. The rest of the message is telling you that Ivy is downloading the required libraries and caching them. The default cache location is a

Listing 5

```
<project name = "IvyAnt"  basedir="."
    default = "compile"
    xmlns:ivy="antlib:org.apache.ivy.ant">

  <property name = "src.dir"
     value="${basedir}/src" />
  <property name = "bin.dir"
     value="${basedir}/bin" />

  <target name = "init">
    <mkdir dir = "${bin.dir}"/>
    <ivy:retrieve />
  </target>

  <target name = "compile" depends= "init">
    <javac srcdir="${src.dir}"
       destdir="${bin.dir}" fork="true">
    </javac>
  </target>
</project>
```

Listing 6

```
<project name = "IvyAnt"  basedir="."
   default = "compile"
   xmlns:ivy="antlib:org.apache.ivy.ant">
  <property name = "src.dir"
     value="${basedir}/src" />
  <property name = "bin.dir"
     value="${basedir}/bin" />
  <property name = "ivy.lib.dir"
     value="${basedir}/lib" />

  <path id="lib.path.id">
    <fileset dir="${ivy.lib.dir}" />
  </path>

  <target name = "init">
    <mkdir dir = "${bin.dir}"/>
    <ivy:retrieve />
  </target>

  <target name = "compile" depends= "init">
    <javac srcdir="${src.dir}"
       destdir="${bin.dir}" fork="true">
      <classpath refid = "lib.path.id"/>
    </javac>
  </target>
</project>
```

directory called `.ivy2/cache` below the user's home directory. The location Ivy uses as the user's home can be changed by setting the **ivy.default.ivy.user.dir** property in the `build.xml` file.

You should also see that a new directory called `lib` has been created, which contains the **apache.commons.lang jar** files. The location and name of this directory can be changed by setting the **ivy.lib.dir** Ant property. Even if you use the default **ivy.lib.dir** value, you must declare it as a property in `build.xml` so that the Java compiler knows where to find the jars.

Even though Ivy has downloaded and cached the dependency and even copied it to a local location within the project, the Java compiler needs to know where to look for the jar files. This is achieved by setting up a path id and using it as the Java compiler's class path (Listing 6).

When you run the Ant script again you will see that Ivy recognises that it already has the dependencies cached, so it does not download them again.

Writing the Ant task to run the application is very simple also. All you need is a path id so that **java** can find the compiled class files and jar files and a **java** task (Listing 7).

Running Ant one last time will give the output **"Hello, World!"**.

As your application grows all you need to do is search for the dependencies you need in `mvnrepository.com`, add them to `ivy.xml` and run Ant to retrieve them. Some dependencies, such as the Microsoft SQL Server driver for Java are not in the Maven repository and it is advantageous to set-up a local repository to host these types of dependencies. I'll describe how to do this in a later article.

It's worth making sure you are clear on the difference between a cache and a repository. A cache is usually local. When you do a build, Ivy checks the cache to see if you already have the required dependencies. If you do, it uses them, otherwise it looks in the repository for them and downloads them. Repositories can be local, but tend to be remote on the internet or on a central server in an organisation. Maven is a repository and stores a large number of libraries.

Using Ivy with Eclipse

Ivy is great if you're using Ant and the command line, but what if you do most of your development in Eclipse and you do not want to use Ant? IvyDe is a new and relatively immature plugin. It will resolve and add the dependencies declared in an `ivy.xml` file to the classpath of your Eclipse

Listing 7

```
<project name = "IvyAnt"  basedir="."
   default = "run"
   xmlns:ivy="antlib:org.apache.ivy.ant">
  <property name = "src.dir"
     value="${basedir}/src" />
  <property name = "bin.dir"
     value="${basedir}/bin" />
  <property name = "ivy.lib.dir"
     value="${basedir}\lib" />

  <path id="lib.path.id">
    <fileset dir="${ivy.lib.dir}" />
  </path>
  <path id="run.path.id">
    <path refid="lib.path.id" />
    <path location="${bin.dir}" />
  </path>

  <target name = "init">
    <mkdir dir = "${bin.dir}"/>
    <ivy:retrieve />
  </target>
  <target name = "compile" depends= "init">
    <javac srcdir="${src.dir}"
       destdir="${bin.dir}" fork="true">
      <classpath refid = "lib.path.id"/>
    </javac>
  </target>
  <target name = "run" depends= "compile">
    <java classpathref="run.path.id"
       classname="IvyAnt"/>
  </target>
</project>
```

project. It is also an editor for `ivy.xml` files with auto completion. The download instructions are the same as for most Eclipse plugins and are given on the IvyDE website [12].

As with Ivy itself, the easiest way to demonstrate the use of IvyDE with Eclipse is with a simple application that has a single dependency:

```
import org.apache.commons.lang.WordUtils;
public class IvyEclipse
{
  public static void main(String[] args)
  {
    final String msg = "hello, world!";
    System.out.println(
       WordUtils.capitalizeFully(msg));
  }
}
```

If you create a Java project called **`IvyEclipse`** and add the class above, Eclipse should complain that it cannot resolve the import **`org.apache`** or **`WordUtils`**. The solution is to add an `ivy.xml` file. To do this:

1. Right click on the project and go to **New→Other**
2. Expand **IvyDE** and select **Ivy File**.
3. Click **Next**
4. Click the **Browser** button next to the container edit box and select the project.
5. Enter an organisation (e.g. Purple Tube) and use the project name as the **Module**.
6. Click **Finish**.

You should be presented with the following `ivy.xml` file (Listing 8).

Add the dependencies from the `ivy.xml` in the Ant example above and save (Listing 9).

To get Ivy to resolve the dependencies, right click on the `ivy.xml` file and select Add Ivy Library. Just click Finish on the IvyDE Managed

Listing 8

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ivy-module version="2.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance" xsi:noNamespaceSchemaLocation=
    "http://ant.apache.org/ivy/schemas/ivy.xsd">
  <info
    organisation="Purple Tube"
    module="IvyAnt"
    status="integration">
  </info>
</ivy-module>
```

Listing 9

```
<ivy-module version="1.0">
  <info organisation="Purple Tube"
     module="IvyEclipse" status="integration"/>
  <dependencies>
    <dependency org="commons-lang"
       name="commons-lang" rev="2.0"
       conf="default"/>
  </dependencies>
</ivy-module>
```

Libraries dialog box for the time being. Then a new element will appear in the project tree called **ivy.xml [*]** and the project errors will be resolved. Expand the element to see that **`org.apache.common.lang`** is being referenced in the Ivy cache.

Unlike Ivy, IvyDE does not copy the dependencies to the project. If you require this behaviour, for example if you need to copy dependent jar files to a web application's WAR directory, you need to use IvyDE in conjunction with Ivy and Ant and set the **`ivy.lib.dir`** property appropriately. Ivy Ant tasks and IvyDE will happily share the same `ivy.xml`.

When you modify `ivy.xml` and save it or check the project out of a source control system, IvyDE should resolve automatically. However, if it does not, right click on **ivy.xml [*]** and select **Resolve**.

Conclusion

Ivy itself is a well featured, immediately useful tool. It has allowed my team to significantly reduce the amount of space used in our subversion repository, while remaining in complete control of our dependencies. In this article I have only scratched the surface of what it can do. I am intending to explore its capabilities further in future articles.

IvyDE still feels very new and needs some work. It is mostly a convenience, replacing the need to run ant to resolve dependencies. I am hoping it's going to come along soon. For now I am tempted to start trying to add features myself. ■

References

[1] Google Web Tool Kit:

[2] Spring Framework: http://www.springsource.org/

[3] Hibernate Object Resource Mapper: https://www.hibernate.org/

[4] Svn Externals: http://svnbook.red-bean.com/en/1.0/ch07s03.html

[5] Maven Software Project Management: http://maven.apache.org/

[6] Ivy, The Agile Dependency Manager: http://ant.apache.org/ivy/

[7] Ant: http://ant.apache.org/

[8] Ivy Eclipse Plugin: http://ant.apache.org/ivy/ivyde/

[9] Eclipse IDE: http://www.eclipse.org/

[10] Ivy Quick Start Guide: http://ant.apache.org/ivy/history/2.1.0-rc1/tutorial/start.html

[11] Maven Repository: http://mvnrepository.com/

[12] IvyDE Download Instructions: http://ant.apache.org/ivy/ivyde/download.cgi

Hunting the Snark (Part 4)

Alan Lenton investigates engineering in software.

Manager: 'I want some serious answers. How long will this program take to make?'

Programmer: '45 seconds give or take 5 seconds.'

Manager: 'Don't be ridiculous!'

And yet it's actually true. Before I justify this statement, lets step back a bit and take a wider look at what provoked this somewhat contentious idea.

I was perusing the ACCU General mailing list, as one does, when someone asked whether people thought software engineering was a true branch of engineering. The question flowed out of a discussion on what the use of membership of the British Computer Society (BCS) was. There was quite a lot of discussion on the issue – hotly debated different views and the usual multiple branches off into different topics. One of the great things about ACCU general is that you never know who is going to hijack your thread and take it somewhere you never thought of. (Hi, Paul!)

This got me thinking and doing a bit of research on the web. The first thing I had a look at was the latest 44 page magnum opus entitled *Engineering Values in IT* from the BCS, Royal Academy of Engineering, and the Institution of Engineering and Technology [1]. Predictably, the report's conclusions fell into the programming is now mature, therefore it should be a profession, and we should have its practitioners required to have professional chartered status.

To quote from the document summary:

> The concerns of this report relate to the area of IT dealing with the development of software-based IT systems. This area of IT has benefited greatly from the input of engineering methods and skills. It is the argument of this report that IT systems engineering has now reached the point of maturity where it is a codified discipline that must be recognised as a required capability for system developers. Like all branches of engineering, IT systems engineering has a basis in scientific theory.

I think they should give a citation to the Wizard of Oz who first pointed out that you don't need a brain, you only need a diploma... [2]

The problem is that, among other things, the idea that the 'maturity' of a discipline is a sufficient condition for declaring it a profession requiring the membership of a professional body to practise is unproven. The scientific basis of software engineering is also a matter of opinion. I posted to ACCU General, pointing out that the engineering disciplines are based on physical laws, and asking what physical laws programming was based on. No one came up with an answer. Neither did the BCS *et al* document.

So what else is out there?

Well, as it happens, the well respected Tom DeMarco, inventor of the term software engineering, has recently written a redux on this very issue. In a short and pithy piece in *Viewpoints* entitled 'Software Engineering: An idea whose time has come and gone?' [3]. It's only two pages, but it takes a sharp look at his 1982 book on software metrics [4], and comes to the conclusion that he was wrong. Wrong because you can't measure everything, and wrong because you *can* control what you can't measure. Though, of course, it's more difficult!

Interestingly, he points out that it makes excellent sense to engineer software, but that isn't what 'software engineering' has come to mean. He also looks at controlling software in terms of its Return on Investment (RoI). To use his example, consider two projects. Project A will eventually cost about a million dollars and produce value of around $1.1 million. Project B will eventually cost about one million dollars and will eventually produce value of around $50 million. Clearly the most important thing for project A is controlling the costs, or it will make a loss. The most important thing for project B is making sure that it realises its potential!

Actually, the real question we need to ask is, 'Why on earth are we wasting time on so many projects like project A that only produce marginal value?'

Moving on, the next piece of interesting writing on the topic I found was a paper by Chuck Connell in *Dr.Dobbs*, entitled 'Software Engineering != Computer Science' [5]. I guess the title says it all. Connell draws a line. Below it are things like complexity, algorithms, cryptography, network analysis, and queuing theory. This he refers to as 'clean', meaning that they are formal mathematical entities which can be proven.

Above the line, which he refers to as the 'bright line', are things like design patterns, architectural style, estimation, portability, testability, and maintainability (I would also add usability to his list). These parts he argues are inherently messy, and the reason is that they involve humans. He proposes a thesis – Connell's Thesis: Software engineering will never be a rigorous discipline with proven results, because it involves human activity.

Hmm. I'm not convinced. Are regular engineers not human? Do humans not use the product of the engineering discipline? Connell is, in my view definitely on to something, but, I don't think his conclusions are correct. (Read his paper, it's only three pages, easy to read and well argued. You may feel I'm doing him an injustice.)

He's clearly correct that software engineering falls into two categories though – a computing-science base and 'the rest'. But why is that? We will pursue this question further in the next issue. ■

References

[1] http://www.raeng.org.uk/news/publications/list/reports/Engineering_values_in_IT.pdf

[2] http://www.filmsite.org/wiza5.html

[3] http://www2.computer.org/cms/Computer.org/ComputingNow/homepage/2009/0709/rW_SO_Viewpoints.pdf

[4] *Controlling Software Projects: Management, Measurement, and Estimation* by Tom DeMarco (Prentice Hall/Yourdon Press, 1982)

[5] http://www.ddj.com/architect/217701907

ALAN LENTON

Alan is a programmer, a sociologist, a games designer, a wargamer, writer of a weekly tech news and analysis column, and an ocassional writer of short stories (see http://www.ibgames.com/alan/crystalfalls/index.html if you like horror). None of these skills seem to be appreciated by putative employers...

What is C++0x?

Bjarne Stroustrup examines some of the changes to the C++ language for C++0x.

This paper has been split into two for publication in CVu. In this first installment, Bjarne Stroustrup examines some of the changes to the core language of C++ for the up-coming C++0x Standard.

This paper illustrates the power of C++ through some simple examples of C++0x code presented in the context of their role in C++. My aim is to give an idea of the breadth of the facilities and an understanding of the aims of C++, rather than offering an in-depth understanding of any individual feature. The list of language features and standard library facilities described is too long to mention here, but a major theme is the role of features as building blocks for elegant and efficient software, especially software infrastructure components. The emphasis is on C++'s facilities for building lightweight abstractions.

Introduction

What is C++0x? I don't mean 'How does C++0x differ from C++98?' I mean what kind of language is C++0x? What kinds of programming are C++0x good for? In fact, I could drop the '0x' right here and attack the fundamental question directly: What is C++? Or, if you feel pedantic, 'What will C++ be once we have the facilities offered by the upcoming standard?' This is not an innocent 'just philosophical' question. Consider how many programmers have been harmed by believing that 'C++ is an object-oriented language' and 'religiously' built all code into huge class hierarchies, missing out on simpler, more modular approaches and on generic programming. Of course C++ also supports OOP, but a simple label – especially if supported by doctrinaire teaching – can do harm by overly narrowing a programmer's view of what can be considered reasonable code.

I will discuss the 'what is?' question in the context of C++0x, the upcoming revision of the ISO C++ standard. The examples will emphasize what's new in C++0x, but the 'commentary' will emphasize their place in the general C++ design philosophy. Despite 'x' in 'C++0x' becoming hexadecimal, I'll use the present tense because most features described are already available for exploratory or experimental use (if not necessarily all in the same implementation). Also, for the benefit of people with an allergy to philosophy, I will discuss ideas and facilities in the context of concrete code examples.

The rest of this paper is organized like this:

- Simplifying simple tasks
- Initialization
- Support for low-level programming
- Tools for writing classes
- Concurrency
- The sum of the language extensions
- Standard library components
- So, what does this add up to?

BJARNE STROUSTRUP

Bjarne Stroustrup designed and implemented the C++ programming language. He can be contacted at www.research.att.com/~bs

That is, I roughly proceed from the simple to the complex. If you want more examples, more details, and/or more precision, see my online C++0x FAQ [1], which contains references to the actual proposals with the names of the people who did most of the work and to the draft standard itself [2].

Simplifying simple tasks

Much of what there is to like and dislike about a programming language are minor details; the kind of minor design decisions, omissions, and mistakes that escape academic attention. For C++0x, quite a bit of attention has been paid to 'removing embarrassments' found in C++98 and simplifying and generalizing rules for the use of the language and standard library.

Deducing a type, etc.

Consider:

```
cout << sqrt(2);
```

Obviously, this should work, but it does not in C++98: The compiler couldn't decide which of the several overloads of **sqrt()** to pick. Now it can (there are now sufficient overloads to get a right answer), but what if you want the value 'right here' and not as an argument to an overloaded function or a template?

```
auto x = sqrt(2);
```

By using **auto** instead of a specific type, you tell the compiler that the type of **x** is the type of its initializer. In this case, that happens to be **double**. In many templates, **auto** can save you from over-specification and in some cases **auto** saves you from writing some long-winded type:

```
for (
  auto p = v.begin(); p!=v.end(); ++p) cout <<*p;
```

Say that **v** was a **vector<list<double,Myallocator<double>>>** and that a suitable definition of **operator<<()** for lists was available. In this case, I saved myself from writing something like

```
for (
   vector<list<double,
   Myallocator<double>>>::iterator p = v.begin();
 p!=v.end();
 ++p)
   cout <<*p;
```

I consider that improvement significant – even more so for a reader and a maintainer than for the original programmer.

auto is the oldest improvement in C++0x: I first implemented it in C with Classes in the winter of 1983/84, but was forced to take it out because of the obvious(?) C incompatibility. Please also note that I did not say **vector< list< double, Myallocator<double> > >**. We can now do without those annoying extra spaces.

Such 'trivial improvements' can matter disproportionally. A tiny stone in a boot can spoil a whole day and a tiny irregularity in a language can make a programmer's day miserable. However, a programmer cannot simply remove a language irregularity – at best it can be hidden behind an interface until a standards committee gets around to the problem.

Range for loop

Let's step up one level from the minute to the small. Why did we play around with iterators to print the elements in that vector? We need the

generality offered by iterators for many important loops, but in most cases, we don't care about details, we just want to do something to each element, so we can say:

```
for (auto x : v) cout << x;
```

That's not just shorter, but also a more precise specification of what we do. If we wanted to access neighboring elements, iterate backwards, or whatever, we can do so (using the familiar and general for loop), but with this 'range for loop' we know that nothing complicated happens in the loop body as soon as we see the header of the **for**-statement.

This range-**for** loop works for all ranges; that is, for every data structure that has a beginning and an end so that you can iterate through it in the conventional way. So it works for **std::vector**, **std::list**, **std::array**, built-in arrays, etc.

Note the use of **auto** in that loop; I did not have to mention **v**'s type. That's good because in generic programs that type can be very hard to express. Consider:

```
template<class C> print_all(const C& v)
{
  for (const auto& x : v) cout << x;
}
```

This can be called with any 'container' or 'range' providing a **begin()** and an **end()**. Since **C** was declared **const** and I don't know the size of the elements, I decorated **auto** with **const** and **&**.

Initialization

Trying to make rules more general and to minimize spurious typing has a long tradition in C++ and most other languages. I suspect it is a never-ending task, but of course the desire for uniformity of rules and simple notation is not restricted to trivial examples. In particular, every irregularity is a source of complexity, added learning time, and bugs (when you make the wrong choice among alternatives). Irregularity also becomes a barrier to generic programming, which critically depends on identical notation for a set of types. One area where uniformity and generality is seriously lacking in C++98 is initialization. There is a bewildering variety of syntaxes (**{…}**, **(…)**, **=**..., or default), semantics (e.g., copy or construct), and applicability (can I use a **{…}** initializer for **new**? for a **vector**? Can I use a **(…)** initializer for a local variable?). Some of this irregularity goes all the way back to the early days of C, but C++0x manages to unify and generalize all of these mechanisms: You can use **{…}** initialization everywhere. For example, see Listing 1.

What could be simpler than initializing an **int**? Yet, we see surprises and irregularities. This gets much more interesting when we consider aggregates and containers:

```
int a[] = { 1, 2, 3 };
S s = { 1, 2, 3 };                  // ok, maybe
std::vector<int> v1 = { 1, 2, 3 }; // new in C++0x
std::vector<int> v2 { 1, 2, 3 };   // new in C++0x
```

It always bothered me that we couldn't handle **v1** in C++. That violated the principle that user-defined and built-in types should receive equivalent support (so why does my favorite container, **vector**, get worse support than the perennial problem, array?). It also violated the principle that fundamental operations should receive direct support (what's more fundamental than initialization?).

Listing 1

```
int v1 = 7;
int v2(7);
int v3 = { 7 }; // yes that's legal C and C++98
int v4 {7};     // new for C++0x: initialize v4 with 7

int x1;         // default x1 becomes 0 (unless x1 is a local variable)
int x2();       // oops a function declaration
int x3 = {};    // new for C++0x: give x3 the default int value (0)
int x4{};       // new for C++0x: give x4 the default int value (0)
```

Listing 2

```
int a[] = { 1, 2, 3 };
void f1(const int(&)[]);    // reference to array
f1({1,2,3});                    // new in C++0x
p1 = new a[]{1,2,3};            // new in C++0x

S s = { 1, 2, 3 };              // ok, maybe
void f2(S);
f2({1,2,3});                    // new in C++0x
p2 = new S {1,2,3};             // new in C++0x

std::vector<int> v2 { 1, 2, 3 }; // new in C++0x
void f3(std::vector<int>);
f3({ 1, 2, 3 });                // new in C++0x
p3 = new std::vector{1,2,3};    // new in C++0x
```

When does **S s = { 1, 2, 3 };** actually work? It works in C++98 iff **S** is a **struct** (or an array) with at least three members that can be initialized by an **int** and iff **S** does not declare a constructor. That answer is too long and depends on too many details about **S**. In C++0x, the answer boils down to 'iff **S** can be initialized by three integers'. In particular, that answer does not depend on how that initialization is done (e.g. constructors or not). Also, the answer does not depend on exactly what kind of variable is being initialized. Consider Listing 2.

Why didn't I just say **void f1(int[])**? Well, compatibility is hard to deal with. That **[]** 'decays' to ***** so unfortunately **void f1(int[])** is just another way of saying **void f1(int*)** and we can't initialize a pointer with a list. Array decay is the root of much evil.

I don't have the time or space to go into details, but I hope you get the idea that something pretty dramatic is going on here: We are not just adding yet another mechanism for initialization, C++0x provides a single uniform mechanism for initializing objects of all types. For every type **X**, and wherever an initialization with a value **v** makes sense, we can say **X{v}** and the resulting object of type **X** will have the same value in all cases. For example:

```
X x{v};
X* p = new X{v};
auto x2 = X{v};  // explicit conversion
void f(X);
f(X{v});
f({v});
struct C : X {
  C() : X{v}, a{v} { }
  X a;
  // …
};
```

Have fun imagining how to do that using C++98 for all types you can think of (note that **X** might be a container, **v** might be a list of values or empty). One of my favorites is where **X** is a **char*** and **v** is **7**.

Support for low-level programming

There is a school of thought that all programming these days is 'web services' and that 'efficiency' ceased to matter decades ago. If that's your world, the lower reaches of C++ are unlikely to be of much interest. However, most computers these days are embedded, processors are not getting any faster (in fact many are getting slower as multi-cores become the norm), and all those web-services and high-level applications have to be supported by efficient and compact infrastructure components. So, C++0x has facilities to make C++ a better language for low-level systems programming (and for programming aiming for efficiency in general).

At the bottom of everything in a computer is memory. Working directly with physical memory is unpleasant and error prone. In fact, if it wasn't for the C++0x memory model

(covered in Part 2) it would be impossible for humans to write code at the traditional 'C language level'. Here I'm concerned about how to get from the world of **int**s, arrays, and pointers to the first level of abstraction.

Plain old data and layout

Consider

```
struct S1 {
  int a, b;
};

struct S2 {
  S2(int aa, int bb) : a{aa}, b{bb} { }
  S2() { } // leave a and b uninitialized
  int a,b;
};
```

S1 and **S2** are 'standard-layout classes' with nice guarantees for layout, objects can be copied by **memcpy()** and shared with C code. They can even be initialized by the same **{…}** initializer syntax. In C++98, that wasn't so. They required different initialization syntaxes and only **S1** had the nice guarantees, but **S2** simply wasn't a POD (the C++98 term for 'guaranteed well-behaved layout').

Obviously, this improvement shouldn't be oversold as 'major', but constructors are really nice to have and you could already define ordinary member functions for a POD. This can make a difference when you are close to the hardware and/or need to interoperate with C or Fortran code. For example, **complex<double>** is a standard-layout class in C++0x, but it wasn't in C++98.

Unions

Like **struct**s, C++98 **union**s had restrictions that made them hard to use together with the abstraction mechanisms, making low-level programming unnecessarily tedious. In particular, if a class had a constructor, a destructor, or a user-defined copy operation, it could not be a member of a **union**. Now, I'm no great fan of **union**s because of their misuses in higher-level software, but since we have them in the language and need them for some 'close to the Iron' tasks, the restrictions on them should reflect reality. In particular, there is no problem with a member having a destructor; the real problem is that if it has one it must be invoked iff it is the member used when the union is destroyed (iff the union is ever destroyed). So the C++0x rules for a union are:

- No virtual functions (as ever)
- No references (as ever)
- No bases (as ever)
- If a union has a member with a user-defined constructor, copy, or destructor then that special function is 'deleted;' that is, it cannot be used for an object of the union type.

This implies that we can have a **complex<double>** as a union member. It also implies that if we want to have members with user-defined copy operations and destructors, we have to implement some kind of variant type (or be very careful about how we use the objects of the unions or the compiler will catch us). For example, see Listing 3.

I'm still not a great fan of unions. It often takes too much cleverness to use them right. However, I don't doubt that they have their uses and C++0x serves their users better than C++98.

Did you notice the **enum class** notation above? An **enum class** is an **enum** where the enumerators have class scope and where there is no implicit conversion to **int**. I used a **class enum** (a 'strongly typed enum') here, so that I could use tag names without worrying about name. Also, note the use of **Tag::**, in C++0x we can qualify an enumerator with the name of its enumeration for clarity or disambiguation.

General constant expressions

Compile-time evaluation can be a major saver of run time and space. C++98 offers support for some pretty fancy template meta-programming.

Listing 3

```
class Widget { // 3 alternative implementations
               // represented as a union
private:
  enum class Tag { point, number, text } type;
  // discriminant
  union {      // compact representation
    point p;   // point has constructor
    int i;
    string s;  // string has default constructor,
               // copy operations, and destructor
  };
  // ...
  widget& operator=(const widget& w)
  // necessary because of  the string
  // union member
  {
    if (type==Tag::text && w.type==Tag::text) {
      s = w.s;  // usual string assignment
      return *this;
    }
    if (type==Tag::text) s.~string();
    // destroy (explicitly!)
    switch (type=w.type) {
    case Tag::point: p = w.p; break;
    // normal copy
    case Tag::number: i = w.i; break;
    case Tag::text:new(&s)(w.s); break;
    // placement new
    }
    return *this;
  }
};
```

However, the basic constant expression evaluation facilities are somewhat impoverished: We can do only integer arithmetic, cannot use user-defined types, and can't even call a function. C++0x takes care of that by a mechanism called **constexpr**:

- We can use floating-point in constant expressions
- We can call simple functions ('**constexpr** functions') in constant expressions
- We can use simple user-defined types ('literal types') in constant expressions
- We can request that an expression must be evaluated at compile time

For example, see Listing 4.

Here **constexpr** says that the function must be of a simple form so that it can be evaluated at compile time when given constant expressions as arguments. That 'simple form' is a single return statement, so we don't do loops or declare variables in a **constexpr** function, but since we do have recursion the sky is the limit. For example, I have seen a very useful integer square root **constexpr** function.

Listing 4

```
enum Flags { good=0, fail=1, bad=2, eof=4 };

constexpr int operator|(Flags f1, Flags f2) {
   return Flags(f1|f2); }

void f(Flags x)
{
  switch (x) {
  case bad:     /* ... */ break;
  case eof:     /* ... */ break;
  case bad|eof: /* ... */ break;
  default:      /* ... */ break;
  }
}
```

Listing 5

```
constexpr int x1 = bad|eof;  // ok
void f(Flags f3)
{
  constexpr int x2 = bad|f3; // error: can't evaluate at compile time
  const int x3 = bad|f3;     // ok: but evaluated at run time
  // …
}
```

In addition to being able to evaluate expressions at compile time, we want to be able to require expressions to be evaluated at compile time; **constexpr** in front of a variable definition does that (and implies **const**) – see Listing 5.

Typically we want the compile-time evaluation guarantee for one of two reasons:

- For values we want to use in constant expressions (e.g. case labels, template arguments, or array bounds)
- For variables in namespace scope that we don't want run-time initialized (e.g. because we want to place them in read-only storage).

This also works for objects for which the constructors are simple enough to be **constexpr** and expressions involving such objects (Listing 6).

Who needs this? Why isn't 'good old **const**' good enough? Or conversely, as one academic in all seriousness asked, 'Why don't you just provide a full compile-time interpreter?' I think the answers to the first two questions are interesting and relate to general principles. Think: what do people do when they have only 'good old **const**'? They hit the limits of const and proceed to use macros for examples like the **Flags** one. In other words, they fall back on typeless programming and the most error-prone abstraction mechanism we know. The result is bugs. Similarly, in the absence of compile-time user-defined types, people revert to a pre-classes style of programming which obscures the logic of what they are doing. Again, the result is bugs.

Listing 6

```
struct Point {
  int x,y;
  constexpr Point(int xx, int yy) : x{xx}, y{yy} { }
};
constexpr Point origo { 0,0 };
constexpr int z { origo.x };
constexpr Point a[] { Point{0,0}, Point{1,1}, Point{2,2} };
constexpr int x { a[1].x };  // x becomes 1
```

I observed two other phenomena:

- Some people were willing to go to extremes of cleverness to simulate compile-time expression evaluation: Template instantiation is Turing complete, but you do have to write rather 'interesting' code to take advantage of that in many cases. Like most workarounds, it's much more work (for programmers and compilers) than a specific language feature and with the complexity comes bugs and learning time.
- Some people (mostly in the embedded systems industry) have little patience with Turing completeness or clever programming techniques. On the other hand, they have a serious need to use ROM, so special-purpose, proprietary facilities start to appear.

The **constexpr** design is interesting in that it addresses some serious performance and correctness needs by doing nothing but selectively easing restrictions. There is not a single new semantic rule here: it simply allows more of C++ to be used at compile time.

Narrowing

Maybe you noticed that I used **{}** initialization consistently. Maybe, you also thought that I was uglifying code and was over-enamored by a novel feature? That happens, of course, but I don't think that is the case here. Consider:

```
int x1 = 64000;
int x2 { 64000 };
```

We can have a nice friendly discussion about the aesthetics of those two definitions and you might even point out that the **{}** version requires one more keystroke than the **=** one. However, there is one significant difference between those two forms that makes me choose **{}**. The **{}** version doesn't allow narrowing and I failed to tell you that the two definitions were written for a machine with 16-bit **int**s. That means that the value of **x1** could be very surprising, whereas the definition of **x2** causes a compile-time error.

When you use **{}** initializers, no narrowing conversions are allowed:

- No narrowing integral conversions (e.g., no **int**-to-**char** conversion)
- No floating-to-integral conversions (e.g. no **double**-to-**int** or **double**-to-**bool** conversion)
- When an initializer is a constant expression the actual value is checked, rather than the type (e.g. **char c {'x'};** is ok because **'x'** fits in a **char**)

This directly addresses a source of nasty errors that has persisted since the dawn of C.

Tools for writing classes

Most of what is good, efficient, and effective about C++ involves designing, implementing, and using classes. Thus, anything that is good for classes is good for C++ programmers. People often look for 'major features' and 'solutions', but I think of language features as building blocks: If you have the right set of building blocks, you can apply them in combination to provide 'solutions'. In particular, we need simple and powerful 'building blocks' to provide general or application-specific abstractions in the form of classes and libraries. Thus, 'little features' that are unimportant in themselves may have a major impact as one of a set of interrelated features. When designed well, the sum is indeed greater than the parts. Here, I will just give three examples: initializer-list constructors, inheriting constructors, and move semantics.

Initializer list constructors

How did we manage to get a **std::vector** to accept a list of elements as its initializer? For example:

```
vector<int> v0 {};              // no elements
vector<int> v1 { 1 };           // one element
vector<int> v2 { 1,2 };         // two elements
vector<int> v3 { 1,2,3};        // three elements
vector<int> v4 { 1, 2, 3, 4, 5, a, b, c, d, x+y,
                 y*z, f(1,d) }; // many elements
```

That's done by providing vector with an 'initializer-list constructor' (Listing 7).

Whenever the compiler sees a **{…}** initializer for a vector, it invokes the initializer constructor (only if type checking succeeds, of course). The **initializer_list** class is known to the compiler and whenever we

Listing 7

```
template<class T> class vector {
  vector(std::initializer_list a)
  // initializer-list constructor
  {
    reserve(a.size());
    uninitialized_copy(v.begin(),v.end(),
                       a.begin());
  }
  // …
};
```

Listing 8

```
template<class T> class Vec : public
std::vector<T> {
public:
  using vector<T>::vector;
  // use the constructors from the base class
  T& operator[](size_type i) {
     return this->at(i); }
  const T& operator[](size_type i) const {
     return this->at(i); }
};
```

write a {…} list that list will (if possible) be represented as an **initializer_list** and passed to the user's code. For example:

```
void f(int,initializer_list<int>,int);
f(1,{1,2,3,4,5},1);
```

Of course it is nice to have a simple and type safe replacement for the **stdargs** macros, but the serious point here is that C++ is moving closer to its stated ideal of uniform support of built-in types and user-defined types [3]. In C++98, we could not build a container (e.g., **vector**) that was as convenient to use as a built-in array; now we can.

Inheriting constructors

In C++98, we can derive a class from another, inheriting the members. Unfortunately, we can't inherit the constructors because if a derived class adds members needing construction or virtual functions (requiring a different virtual function table) then its base constructors are simply wrong. This restriction – like most restrictions – can be a real bother; that is, the restriction is a barrier to elegant and effective use of the language. My favorite example is a range checked vector (see Listing 8).

The solution, to provide an explicit way of 'lifting up' constructors from the base, is identical to the way we (even in C++98) bring functions from a base class into the overload set of a derived class (Listing 9).

Thus, this language extension is really a generalization and will in most contexts simplify the learning of use of C++. Note also the use of the {…} initialization mechanism in that last example: Of the alternatives, only **f(complex<double>)** can be initialized with a pair of doubles, so the {…} notation is unambiguous.

Move semantics

Consider

```
template<class T> void swap(T& a, T& b)
{
  T tmp = a;  // copy a into tmp
  a = b;      // copy b into b
  b = tmp;    // copy tmp into b
}
```

But I didn't want to copy anything, I wanted to swap! Why do I have to make three copies? What if **T** is something big and expensive to copy? This is a simple and not unrealistic example of the problem with copy: After each copy operation, we have two copies of the copied value and often that's not really what we wanted because we never again use the original. We need a way of saying that we just want to move a value!

In C++0x, we can define 'move constructors' and 'move assignments' to move rather than copy their argument:

```
template<class T> class vector {
  // ...
  vector(const vector&);      // copy constructor
  vector(vector&&);           // move constructor
  vector& operator=(const vector&);
                              // copy assignment
  vector& operator=(vector&&);// move assignment
};
```

The **&&** means 'rvalue reference'. An rvalue reference is a reference that can bind to an rvalue. The point about an rvalue here is that we may assume that it will never be used again after we are finished with it. The obvious implementation is for the **vector** move constructor therefore to grab the representation of its source and to leave its source as the empty **vector**. That is often far more efficient than making a copy. When there is a choice, that is, if we try to initialize or assign from an rvalue, the move constructor (or move assignment) is preferred over the copy constructor (or copy assignment). For example, see Listing 10.

Obviously (once you become used to thinking about moves), no **vector** is copied in this example. If you have ever wanted to efficiently return a large object from a function without messing with free store management, you see the point.

So what about **swap()**? The C++0x standard library provides:

```
template<class T>
void swap(T& a, T& b) // "perfect swap" (almost)
{
  T tmp = std::move(a);
  a = std::move(b);
  b = std::move(tmp);
}
```

The standard library function **move(x)** means 'you may treat **x** as an rvalue'. Rvalue references can also be used to provide perfect forwarding: A template function **std::forward()** supports that. Consider a 'factory function' that given an argument has to create an object of a type and return a **unique_ptr** to the created object:

```
template <class T,
   class A1> std::unique_ptr<T> factory(A1&& a1)
{
  return std::unique_ptr<T>{
     new T{std::forward<A1>{a1}}};
}
unique_ptr p1 { factory<vector<string>>{100}) }
```

A **unique_ptr** is a standard-library 'smart pointer' that can be used to represent exclusive ownership; it is a superior alternative to **std::shared_ptr** in many (most?) cases.

One way to look at rvalue references is that C++ had a pressing need to support move semantics: Getting large values out of functions was at best inelegant or inefficient, transferring ownership by using shared pointers was logically wrong, and some forms of generic programming cause people to write a lot of forwarding functions that in principle implies zero run-time costs but in reality were expensive. Sometimes move vs. copy is a simple optimization and sometimes it is a key design decision, so a design

Listing 9

```
struct B {
  void f(int);
  void f(double);
};
struct D : B {
  using B::f;              // "import" f()s from B
  void f(complex<double>);// add an f() to the
                           // overload set
};
D x;
x.f(1);             // B::f(int);
x.f({1.3,3.14});    // D::f(complex<double>);
```

Listing 10

```
vector<int> make_rand(int s)
{
  vector<int> res(s);
  for (auto& x : res) x = rand_int();
  return res;
}
vector<int> v { make_rand(10000) };
void print_rand(int s)
{
  for (auto x : make_rand(s)) cout << x << '\n';
}
```

must support both. The result was rvalue references, which supports the name binding rules that allows us to define **std::move()**, **std::forward()**, and the rules for copy and move constructors and assignments.

User-defined literals

For each built-in type we have corresponding literals:

```
'a'     '\n'            // char
1       345             // int
345u                    // unsigned
1.2f                    // float
1.2     12.345e-7       // double
"Hello, world!"         // C-style string
```

However, C++98 does not offer an equivalent mechanism for user-defined types:

```
1+2i                    // complex
"Really!"s              // std::string
103F   39.5c            // Temperature
123.4567891234df        // decimal floating point
123s                    // seconds
101010111000101b        // binary
12345678901234567890123456789012345678901234567890x
                        // extended-precision
```

Not providing literals for user-defined types is clear violation of the principle that user-defined and built-in types should receive equivalent support. I don't see how C++ could have survived without user-defined literals! Well, I do: inlined constructors are a pretty good substitute and **constexpr** constructors would be even better, but why not simply give people the notation they ask for?

C++0x supports 'user-defined literals' through the notion of literal operators that map literals with a given suffix into a desired type. For example, see Listing 11.

Note the use of **constexpr** to enable compile-time evaluation; **complex<double>** is a literal type. The syntax is **operator""** to say that a literal operator is being defined followed by the name of the function, which is the suffix to be recognized. Given those literal operators, we can write:

```
template<class T> void f(const T&);
f("Hello");         // pass pointer to char*
f("Hello"s);        // pass (5-character)
                    // std::string object
f("Hello\n"s);      // pass (6-character)
                    // std::string object
auto z = 2+3.14i; // 2+complex<double>(0,3.14)
```

The basic (implementation) idea is that after parsing what could be a literal, the compiler always checks for a suffix. The user-defined literal mechanism simply allows the user to specify a new suffix and what is to be done with the literal before it. It is not possible to redefine the meaning of a built-in literal suffix or invent new syntax for literals. In this, user-defined literals are identical to user-defined operators.

A literal operator can request to get its (preceding) literal passed 'cooked' (with the value it would have had if the new suffix hadn't been defined) or 'raw' (the string of characters exactly as typed).

Listing 11

```
constexpr complex<double> operator"" i(
   long double d)     // imaginary literal
{
  return {0,d};       // complex is a literal type
}
std::string operator "" s(const char* p,
   size_t n)          // std::string literal
{
  return string{p,n};// requires free store
                     // allocation
}
```

To get an 'uncooked' string, simply request a **const char*** argument:

```
Bignum operator"" x(const char* p)
{
  return Bignum(p);
}
void f(Bignum);
f(12345678901234567890123456789012345678901234567890x);
```

Here the C-style string

```
"12345678901234567890123456789012345678901234567890"
```

is passed to **operator"" x()**. Note that we did not have to explicitly put those digits into a string, though we could have:

```
f("12345678901234567890123456789012345678901234567890"x);
```

Note that 'literal' does not mean 'efficient' or 'compile-time evaluated'. If you need run-time performance, you can design for that, but 'user-defined literals' is – in the best C++ tradition – a very general mechanism.

The sum of language extensions

When you add it all up, C++0x offers are many new language facilities. The C++0x FAQ lists them:

- __cplusplus
- alignments
- attributes
- atomic operations
- auto (type deduction from initializer)
- C99 features
- enum class (scoped and strongly typed **enum**s)
- copying and rethrowing exceptions
- constant expressions (generalized and guaranteed; **constexpr**)
- decltype
- default template parameters for functions
- defaulted and deleted functions (control of defaults)
- delegating constructors
- Dynamic Initialization and Destruction with Concurrency
- explicit conversion operators
- extended **friend** syntax
- extended integer types
- extern templates
- for statement; see range **for** statement
- generalized SFINAE rules
- in-class member initializers
- inherited constructors
- initializer lists (uniform and general initialization)
- lambdas
- local classes as template arguments
- long long integers (at least 64 bits)
- memory model
- move semantics; see rvalue references
- Namespace Associations (Strong using)
- Preventing narrowing
- null pointer (**nullptr**)
- PODs (generalized)
- range for statement
- raw string literals
- right-angle brackets
- rvalue references
- static (compile-time) assertions (**static_assert**)

Interpreting Custom Unix Shell Scripts in C

Ian Bruntlett learns how to write an interpreter for a custom script language.

Assuming you are a Unix/Linux user, you will have stumbled across scripts that look like a compiled command but, on further investigation have, as the first line in the file a line beginning with `#!` followed by the name of an interpreter. Something like this:

```
#!/usr/bin/python
print "Example Python Script : Hello World"
```

I stored the above 2 lines of code in a file called `pyhello` and had to set its permissions as 'OK to execute' with the command **`chmod +x pyhello`** – then I ran it with **`./pyhello`** and it displayed the text:

```
ian@Rutherford:~/c$ ./pyhello
Example Python Script : Hello World
```

That's interesting so far – script language writers can now have their languages used to create script files. What's involved? – 1) the script language itself and 2) a script program that begins with **`#!`** and the filename of the script. What if we decide to write our own script language (say... Forth :). How do we get **`#!`** support?

After some experimentation, I created a file, `hashbang.c`. It turns out that everything happens in the parameters passed to the program – in this case **`argc`** and **`argv[]`**. Listing 1 is an example of a script language reacting to **`#!`**.

OK, so now we have a script language of sorts (all it does is display the script file passed to it), how do we run it from a script file. Here is an example stored in the file `runme`

```
#!/home/ian/c/a.out
echo first line of text excluding hashbang
one
two
three
ian@Rutherford
```

The first line should be set to the executable file to act as an interpreter. This is what I got when I ran the script file `runme`:

```
SCRIPT LINE : #!/home/ian/c/a.out
SCRIPT LINE : echo first line of text excluding
  hashbang
SCRIPT LINE : one
SCRIPT LINE : two
SCRIPT LINE : three
```

Well, that's that. I'm hoping to write a Forth interpreter in C++ and so I might write some more articles along the way. ■

IAN BRUNTLETT

On and off, Ian has been programming for some years. He is a volunteer system administrator for a mental health charity called Contact (www.contactmorpeth.org.uk). As part of his work, Ian has compiled a free Software Toolkit (http://contactmorpeth/wikispaces.com/SoftwareToolkit).

What is C++0x? (continued)

- suffix return type syntax (extended function declaration syntax)
- template alias
- template typedef; see template alias
- thread-local storage (**`thread_local`**)
- unicode characters
- Uniform initialization syntax and semantics
- unions (generalized)
- user-defined literals
- variadic templates

Fortunately most are minor and much work has been spent to ensure that they work in combination. The sum is greater than its parts.

Like inheriting constructors, some of the new features address fairly specific and localized problems (e.g. raw literals for simpler expression of regular expressions, **`nullptr`** for people who are upset by **`0`** as the notation for the null pointer, and **`enum class`**es for stronger type-checked enumerations with scoped enumerators). Other features, like general and uniform initialization, aim to support more general programming techniques (e.g. **`decltype`**, variadic templates, and template aliases for the support of generic programming). The most ambitious new support for generic programming, concepts, didn't make it into C++0x (see [4]).

When exploring those new features (my C++0x FAQ is a good starting point), I encourage you to focus on how they work in conjunction with old and new language features and libraries. I think of these language features as 'building bricks' (my home town is about an hour's drive from the Lego factory) and few make much sense when considered in isolation. ■

Acknowledgements

The credit for C++0x goes to the people who worked on it. That primarily means the members of WG21. It would not be sensible to list all who contributed here, but have a look at the references in my C++0x FAQ: There, I take care to list names. Thanks to Steve Love, Alisdair Meredith, and Roger Orr for finding bugs in early drafts of this paper.

Notes and references

[1] Bjarne Stroustrup: *C++0x FAQ*. www.research.att.com/~bs/C++0xFAQ.html

[2] Pete Becker (editor): *Working Draft, Standard for Programming Language C++*. [N2914=09-0104] 2009-06-22. Note: The working paper gets revised after each standards meeting.

[3] Bjarne Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. 1994.

[4] Bjarne Stroustrup: 'The C++0x "Remove concepts" Decision'. *Dr. Dobbs*; July 2009. http://www.ddj.com/cpp/218600111 and *Overload* 92; August 2009.

[5] Bjarne Stroustrup: 'Concepts and the future of C++' interview by Danny Kalev for DevX. 2009. http://www.devx.com/cplus/Article/42448.

In the next issue, Bjarne looks at the new features for supporting concurrency and regular expressions, and how the changes to the C++ Standard Library make use of the new language features to make programming in C++ simpler and more effective than ever.

A Game of Dice

Baron Muncharris sets a challenge.

Greetings Sir R-----! Sit down and take a glass of this rather fine brandy. Now I know that you are a gentleman who finds that a small wager aids tremendously in the appreciation of fine liquor, so let me tell you about a splendid dice game that I learned from the King of the Moon.

It was upon the occasion of my triumphant victory over the invading forces from the moons of Mars. Prospects had been bleak since although, as is well known, Moon men are supreme swordsmen, the invasion took place during the night after the celebration of the King's jubilee and they were, to a man, abed suffering vivid nightmares owing to an excessive consumption of cheese.

So it was that, armed with nothing but my trusty rapier, a good stock of wine and a ha'penny's worth of pepper, I set upon the invading hoard.

But I digress.

As reward for my comprehensive victory the King offered me a number of bottles of Moon wine equal to my score in his ingenious dice game, the rules of which I shall recount forthwith.

I began the game with a score of zero and was directed to roll a die some numerous times. After each roll I was to add the die's score to mine own, but in the event that I rolled a 1, I was to subsequently halve my current score. Upon completion of the game, I was instructed to report to the cellar-master to collect my bounty.

Rather than play for that most fortifying of spirits – for I have unfortunately exhausted said bounty – I propose that we should play for coin. I shall shortly inform you of the number of dice that you must roll and the stake that I shall require from you to enter the contest. It is for you to determine whether or not it takes your fancy!

I have been informed by a student of the recently discovered theory of wager that it is possible to determine with certainty the expected winnings of any such game, but as he is of common stock, he is almost certainly not to be trusted.

I propose that we play two rounds of this splendid game of chance.

To commence, I suggest a game of 10 dice for a stake of 23¾. Is this to your satisfaction?

To conclude, I suggest a game of 100 dice for a stake of 41. Is this to your liking?

Now be a good fellow and pass the brandy!

Listing 1

```
size_t
roll()
{
  return size_t(6.0 * double(rand())/
     (double(RAND_MAX)+1.0)) + 1;
}

double
play(const size_t n)
{
  double pot = 0.0;
  for(size_t i=0;i!=n;++i)
  {
    const size_t spots = roll();
    pot += double(spots);
    if(spots==1) pot /= 2.0;
  }
  return pot;
}
```

Listing 1 is a C++ implementation of the game.

Desert Island Books

Paul Grenyer introduces Jon Jagger.

I've been doing this series for ten months now and every person has been brilliant. Jon's Desert Island Books is the longest yet and, I have to admit, the one I've enjoyed the most so far. As I know I've said to many people, and maybe even in a previous Desert Island Books, this series isn't really about books, it's about people and Jon has demonstrated that admirably. I've known Jon for a few years and I had no idea he liked fishing!

For me, Jon has always been up there with Kevlin as the best of the best of us. He not only understands how to write software, but he understands the people that write it and processes. I've known him 'on-list' pretty much since I joined the ACCU and I first met him in person, like so many others, in the corner of the Dirty Duck in Stratford upon Avon at almost the same time as the famous Phil Hibbs incident. At the conference the following year Jon had a slight problem remembering exactly who I was, but as you can tell I'm not holding a grudge. I've bumped into Jon at a couple of C++ panel meetings also.

Jon has helped me out on numerous occasions, was the reason I bought *The Mythical Man Month* and even took Aeryn with him into a Formula 1 team. If only it could have been Williams!

Jon Jagger

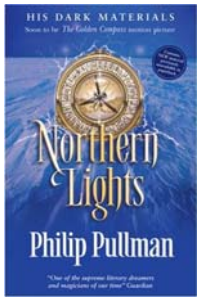
I'll start with my two albums. I considered choosing an album of my own. Not stuff I personally composed or sang. Heaven forbid. I couldn't sing in tune if my life depended on it. I mean a compilation of singles from different artists. I bought loads and loads of proper vinyl singles as a boy. The singles I could list. In fact I think I will. Pad it out a bit... What comes to mind.... Early OMD stuff such as 'Electricity'. 'This is the day' by The The. What a cracking single that is. 'Echo Beach' by Martha and the Muffins. I liked a lot of the mod stuff too. 'Poison Ivy' by the Lambrettas was a favourite. Pretty much anything from early Dexys Midnight Runners. Everything by David Bowie. Recently, at my daughter's school concert a very talented fifth form girl played the piano and sang 'True Colours' by Cyndi Lauper, which reminded me what a great song that is. Another favourite from later on was 'Jeans Not Happening' by The Pale Fountains. I bought plenty of singles from previous decades too. Del Shannon, The Isley Brothers, Roy Orbison. 'The House of the Rising Sun' by the Animals. That would definitely be on the compilation album. How many singles could you fit on a compilation CD? A hundred easily. We may be here some time... Lots by the Beach Boys. Louis Armstrong's 'We Have All the Time in the World' (recorded for the James Bond film – *On Her Majesty's Secret Service* – in one take. He died shortly afterwards). In short I like most things with a good tune where I have a reasonable chance of discerning the lyrics by listening (the exception being Kevin Rowland from Dexys). But I decided that would be cheating. So I've mentioned it so I can discount it but at least it gets mentioned. That seems to be a standard tactic employed by previous Desert Island visitors. Instead I've opted for two regular albums instead. My first is *Ziggy Stardust and the Spiders from Mars* by David Bowie. I just love this album. All of this album. Every single track. Every second of every track. It's hard to explain why you love an album and I can't really explain it. Perhaps its not something you should even try to explain. How can you explain how you feel about an album you love and know intimately? Other than to say you love it. And know it intimately. So I'll leave it at that. The second album would have to be the aptly titled *OK COMPUTER* by Radiohead. You'll no doubt be familiar with the haunting 'No Surprises', which is from this album.

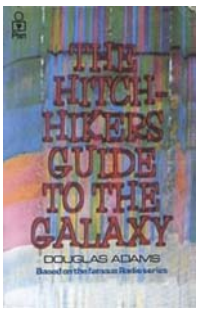
Or maybe Karma Police – 'Arrest this girl, her Hitler hairdo is making me feel ill, and we have crashed her party'. What a great lyric. But neither of these is my favourite. My favourite is all 6 minutes and 23 seconds of 'Paranoid Android'. I tweeted that I was listening to this about a week ago! In fact I've been fishing for the last two days on the River Wye so I haven't heard it for a while (I never take anything electrical with me when fishing – that would just not be right) so excuse me while I listen to it again for a moment... Rain down on me...from a great height...God loves his children...yeah...

So on to some books. Five books. I'm going to try and make choices that haven't been made by previous visitors.

The first is *World Class Match Fishing* [1], which I confidently predict no one will have heard of, let alone read. Fishing is similar to developing software in that both involve large amounts of invisibility. But even so, unless you are keen on freshwater fishing (as I am) I don't recommend it. To explain a little (there is a point honest), match fishing is where a group of fisherman compete against each other to see who can catch the most fish (by weight) in a fixed interval of time (usually 10am–3pm when they're the hardest to catch). Small stretches of a river (or pond/lake) are numbered and marked and the anglers draw a number to determine where to fish. Fish, like people, do not spread themselves out evenly. Quite the opposite. Consequently the vast majority of numbers (they're called swims or pegs) have no chance of winning. It's crazy really. (What has an IQ of 100? Answer 100 match fisherman!) Swims are split into sections; in a match of 60 anglers there might be 5 sections of 12 and there are smaller cash prizes for winning your section (it helps to maintain interest since, as I said, most swims have zero chance of winning). The best anglers will regularly win their sections. Kevin Ashurst was an exceptional fisherman once winning the World Championships. In this book he explains, often in great detail, the thinking behind his tactics and strategies when trying to win. I love it for the unselfish explanation of his secrets but mostly for the insight into his clarity of thought. Good thinking is pretty rare but he had it in abundance. There are some lovely examples of the Lean principle of removing waste. To give you one example – on a river you can sometimes beat everyone in your section even if they are 'better' fishermen than you simply by making sure your float is in the water longer than theirs! How can you do that? One way is simply to slow the float down (that way you spend less time retrieving your float or casting it back out – times when you definitely won't catch a fish).

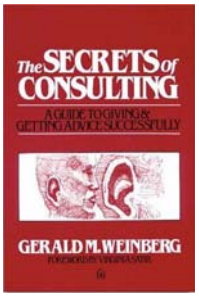
For a novel I'll pick the last one I read that I could not put down once I'd started: *The Northern Lights* [2]. A wonderful book, the first in *His Dark Materials* trilogy, it's a hypnotic mix of fantasy and reality where the fantasy is a grown-up fantasy weaving a rich picture of a parallel but definitely alternate universe with wonderful flights of imagination. The author says he prefers not to explain the meaning of what he writes instead letting the reader draw their own more personal meaning. I guess that means I shouldn't really explain the meaning I draw from it either since it would spoil your enjoyment if you decide to read it. Let's just say that the trilogy has a fairly strong and obvious atheist

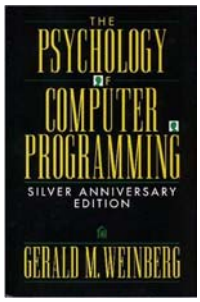




aspect to it. *Hitch Hikers Guide to the Galaxy* [3] was a close second for the novel but I've read that so many times my copy is falling apart.

For my third book I'm going to pick something closer to home. *The Secrets of Consulting* [4]. I have a lot of books by Jerry Weinberg. His most famous is probably *The Psychology of Computer Programming* [5], but some parts of that haven't aged particularly well. He writes that he regrets using the word psychology in its title since he is not, nor has he ever been, a psychologist. Perhaps he doesn't have a certificate or official qualification, but it's clear he has a great understanding of people, of the systems they are involved in, and in consulting – the art of influencing people at their request. This is my favourite Weinberg book by quite some margin and, reading between the lines, I think it is his too. I recall Kent Beck once saying he was greatly influenced by it. It's stuffed full of advice under the banner of consulting, but in truth much of the book has a much broader application. A lot of the book is about change, which is pretty universal. It's also one of those rare books that has quality in depth. The more you read it, the more you see its hidden layers and the deeper your understanding becomes. It's well written and the advice is summarised into numerous pithy laws/aphorisms such as 'The Fast Food Fallacy' (no difference plus no difference plus no difference plus … eventually equals a clear difference, p.173). For a software example of that, consider compiler warnings. A rare gem.





It seems a shame that Desert Island visitors can't choose a couple of their favourite films. Perhaps Paul could add it as a new category. Meanwhile I'm going to have to cheat by including a film screenplay as my fourth book. *The Life of Brian* [6]. Most pythonistas agree this is their finest film. The others are a bit patchy in places but every scene of Brian is a sure fire rib tickler. It's easy forget the controversy it caused when it was released.

The back of the screenplay contains some reviews and, possibly uniquely, two are distinctly unfavourable! Entirely deliberate, of course. I like the New Statesman's review 'Hurray for blasphemy'. And let's not forget the classic song 'Always Look on the Bright Side of Life'. That would be sage advice if you were stranded on a desert island.

What's it all about?

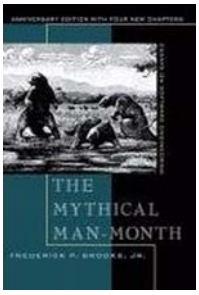
Desert Island Disks is one of BBC Radio 4's most popular and enduring programmes:

http://www.bbc.co.uk/radio4/factual/desertislanddiscs.shtml

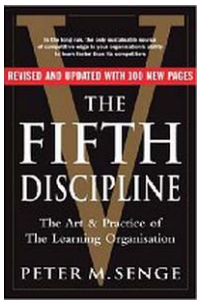
The format is simple: each week a guest is invited to choose the eight records they would take with them to a desert island.

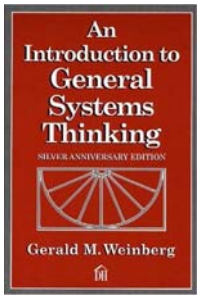
I've been thinking for a while that it would be entertaining to get ACCU members to choose their Desert Island Books. The format will be slightly different from the Radio 4 show. Members will choose about 5 books, one of which must be a novel, and up to two albums. The programming books must have made a big impact on their programming life or be ones that they would take to a desert island. The inclusion of a novel and a couple of albums will also help us to learn a little more about the person. The ACCU has some amazing personalities and I'm sure we only scratch the surface most of the time.

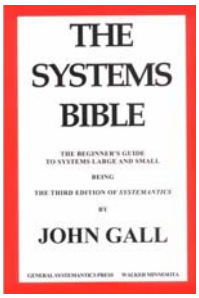
Each issue of CVu will have someone different. If you would like to share your Desert Island Books please email me: paul.grenyer@gmail.com.



My last choice is proving difficult. Lots of worthy books will miss out. Classics such as *Programming Pearls* [7] and *The Mythical Man Month* [8]. I've been reading a lot about Systems Thinking recently. Systems Thinking essentially means non-linear thinking. Human beings have evolved a very strong association that cause and effect are simple and linear; that cause and effect are local in space and time. Unfortunately the world of software is not as simple as that. Developing software takes time. Another phrase sometimes used in this context is 'dynamic complexity' a term from *The Fifth Discipline* [9], where Peter Senge draws a useful distinction between detail complexity and dynamic complexity. The excellent *An Introduction to General Systems Thinking* [10] is recommended and almost got the fifth spot, but in the end I've plumped for *The Systems Bible* [11]. This is a light hearted, slightly tongue-in-cheek book with pithy summaries in the forms of Laws and Principles. For example Le Chatelier's Principle 'Systems tend to oppose their own proper function'. Another one is called The Basic Axiom of Systems-function 'Big systems either work on their own or they don't. If they don't you can't make them.' Very apt. It's occasionally laugh-out-loud funny too. Ro/Rs on page 47 for example. Well worth considering if you want to edge away from technical aspects of work for a while.







References

[1] Kevin Ashurst. (1977 Long out of print). *World Class Match Fishing*, Cassell, ISBN 0304-297291.

[2] Phillip Pullman. (1995). *The Northern Lights* (*The Golden Compass* in USA, Knopf), Scholastic, ISBN 043995178X

[3] Douglas Adams. (1979). *The Hitch Hikers Guide to the Galaxy*, Pan, ISBN 0330258648

[4] Gerald Weinberg. (1985). *The Secrets of Consulting*, Dorset House, ISBN 0932633013

[5] Gerald Weinberg. (1998). *The Psychology of Computer Programming: Silver Anniversary Edition*, Dorset House, ISBN 0932633420

[6] Monty Python. (2001). *The Life of Brian* (screenplay), Metheun, ISBN 0413741303

[7] Jon Bentley, (1989). *Programming Pearls*, Addison Wesley, ISBN 0201103311

[8] Fred Brooks (1985 2nd edition). *The Mythical Man Month*, Addison Wesley, ISBN 0201835959

[9] Peter Senge (2006 2nd edition). *The Fifth Discipline*, Random House, ISBN 1905211201

[10] Gerald Weinberg (2001). *Introduction to General Systems Thinking* Silver anniversary edition, Dorset House, ISBN 0932633498

[11] John Gall (2002 3rd edition). *The Systems Bible: The Beginner's Guide to Systems Large and Small* , General Systemantics Press, ISBN 0961825170

Next issue: Michael Feathers.

Code Critique Competition 59

Set and collated by Roger Orr.

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

Last issue's code

Can someone please help me to understand why the following program crashes? I tried to fix it by using `strncat` but it still doesn't work.

Last issue's code is shown in Listing 1.

Listing 1

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

int main()
{
  char * home_path = NULL;
  char * fullpath = NULL;
  char * fullfile_directory = NULL;
  char buffer[225];
  FILE *f;
  FILE *readfile;

  home_path = getenv("HOME");
  fullfile_directory = strncat(home_path,
    "/snapshot.html", 255-1);
  printf("Searching: %s\n", getenv("HOME"),
    fullfile_directory, 255-1);
  readfile = fopen(fullfile_directory, "r");
  while (!feof(readfile))
  {
    if (readfile != NULL)
    {
      f = fopen("/tmp/output.log", "a+");
      fgets(buffer, 256, readfile);
      if (strstr(buffer, "<title>"))
      {
        printf("Extracting title\n");
        fprintf(f, "title: %s", buffer);
      }
      else if(strstr(buffer, "<h1>"))
      {
        printf("Extracting heading\n");
        fprintf(f, "Heading: %s", buffer);
      }
    }
  }
  fclose(f);
  fclose(readfile);
  return 0;
}
```


ROGER ORR

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

Critiques

Ian Bruntlett <ian.bruntlett@googlemail.com>

One problem with the program is that the buffer size is sometimes passed as 255, sometimes it is 225.

Change 1 introduced a buffer size that, with a single edit, can be updated without breaking a lot of code that assumes what the buffer size is. This change was supported by changes 3,4,6.

Change 5 was introduced to get the program to fail gracefully if the input file is not present.

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

#define BUFFSIZE 255                          // 1

int main()
{
  char * home_path = NULL;
  char * fullpath = NULL;
  char * fullfile_directory = NULL;
  char buffer[BUFFSIZE];                      // 2
  FILE *f;
  FILE *readfile;
  home_path = getenv("HOME");
  fullfile_directory = strncat(home_path,
     "/snapshot.html", BUFFSIZE-1);           // 3
  printf("Searching: %s\n", getenv("HOME"),
     fullfile_directory, BUFFSIZE-1);         // 4
  readfile = fopen(fullfile_directory, "r");

  if ( readfile == NULL )                     // 5
    {
      printf ("Unable to open %s\n",
         fullfile_directory  );
      return 1;
    }

  while (!feof(readfile))
  {
    if (readfile != NULL)
    {
      f = fopen("/tmp/output.log", "a+");
      fgets(buffer, BUFFSIZE, readfile);      // 6
      if (strstr(buffer, "<title>"))
      {
        printf("Extracting title\n");
        fprintf(f, "title: %s", buffer);
      }
      else if(strstr(buffer, "<h1>"))
      {
        printf("Extracting heading\n");
        fprintf(f, "Heading: %s", buffer);
      }
    }
  }
  fclose(f);
  fclose(readfile);
  return 0;
}
```

Pete Disdale <pete@papadelta.co.uk>

There really are so many problems with this piece of code that it's difficult to know quite where to start. But having said that, the main ones are (in no particular order):

- a lack of understanding of the difference between a **`char *`** and an array of **`char`** (the main reason for the crash, I would say)
- the absence of any error useful checking (or checking in the wrong place)
- opening and not closing a file, caused by
- incorrect nesting of code blocks
- errors caused by (probably) using 'magic' numbers for (I presume) buffer sizes, and copy & paste editing with no check afterwards for correct or appropriate function parameters.There being so many 'inelegancies' with this code, the best way I can think of critiquing it is to reproduce the original code and add comments where appropriate, and then to offer a version which, whilst far from perfect addresses these problems and perhaps more importantly does not crash. Well, not here anyway...

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

int main()
{
  char * home_path = NULL;
  char * fullpath = NULL;
```

`fullpath` is never used, though would to my mind be a better choice than **`fullfile_directory`**, though I've left it be.

```
  char * fullfile_directory = NULL;
  char buffer[225];
```

This looks like a typo for 255? One of the dangers in using 'magic' numbers mid-code.

```
  FILE *f;
  FILE *readfile;
  home_path = getenv("HOME");
  fullfile_directory = strncat(home_path,
    "/snapshot.html", 255-1);
```

This is a major problem. Not only is there no check for **`getenv()`** returning **`NULL`**, but the pointer returned could be to a static buffer anywhere. Appending to this is almost guaranteed to overwrite something else important! And to make matters worse, **`strncat()`** will likely not null-terminate the result. **`home_path`** needs to be copied to its own buffer, which should be large enough to hold both **`home_path`** and **`"/snapshot.html"`** plus the null terminator. This buffer could be an array of size **`MAXPATH`**, or **`malloc()`**'d once the size is known. As for the **`255-1`**, ...

```
  printf("Searching: %s\n", getenv("HOME"),
    fullfile_directory, 255-1);
```

Looks like a 'copy&paste from previous line' error. There is a single **`%s`** in the format, yet 3 parameters, and all it would (might) print is what had already been assigned to **`home_path`** earlier.

```
  readfile = fopen(fullfile_directory, "r");
```

No check for **`fopen()`** failing here, yet **`readfile`** is used as valid in the line immediately below...

```
  while (!feof(readfile))
  {
    if (readfile != NULL)
```

... and yet checked here, rather late in the day!

```
    {
      f = fopen("/tmp/output.log", "a+");
```

No check for **`fopen()`** failing here, but as important is that here is not the place to **`fopen()`** the log file anyway: it will be **`fopen()`**'d anew for each input line, and only the last instance is closed. The output log should be opened once, before the read loop, and closed once afterwards. Or at the very least, closed after each **`fopen()`**, inefficient though that would be.

```
    fgets(buffer, 256, readfile);
```

No check for **`gets()`** returning **`NULL`**, which it could do even if the **`feof()`** allowed the loop to be entered. **`fgets()`** should ideally be the loop control mechanism. Also, it is implicitly assumed that all input lines will fit into the 256 byte buffer (254 chars plus a '\n' and a '\0'). If any lines are longer than this, some mechanism must be employed to look for the '\n' and thereby determine whether or not a complete line of input was read.

```
      if (strstr(buffer, "<title>"))
      {
        printf("Extracting title\n");
        fprintf(f, "title: %s", buffer);
```

No check for **`fprintf()`** succeeding/failing, both here and in the **`else`** block below. Also **`strstr()`** is case-sensitive, which might or might not be what is needed for `snapshot.html`

```
      }
      else if (strstr(buffer, "<h1>"))
      {
        printf("Extracting heading\n");
        fprintf(f, "Heading: %s", buffer);
      }
    }
  }
  fclose(f);
```

This **`fclose(f)`** is misplaced relative to its **`fopen()`** as noted above. Also (nitpicking) the return value from **`fclose()`** is unchecked, though at this point in program execution it's moot what one might do about it.

```
  fclose(readfile);
  return 0;
}
```

Success? Perhaps the return code could better reflect the program's exit status?

And here is the alternative version:

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

#define BUFLEN  256  /* or whatever... */
#define SNAPSHOT_FILE  "snapshot.html"
#define LOG_FILE  "/tmp/output.log"

/*
 * These might be overkill for the purposes of
 * the original programmer, but specified in
 * case error cause is required.
```

```
 */
#define HOME_ERROR   1  /* HOME not found */
#define MEM_ERROR    2  /* malloc() error */
#define FOPEN_ERROR  3  /* fopen() error */
#define FREAD_ERROR  4  /* file read error */
#define FWRITE_ERROR 5  /* file write error */

/* function prototypes */
int  main (void);

int main()
{
  char * home_path = NULL;
  char * fullpath = NULL;
    /* unused in original code */
  char * fullfile_directory = NULL;
    /* this is unused here */
  char buffer[BUFLEN];
  FILE *f;
  FILE *readfile;
  int rc = 0;
   /* return code, default to zero */

  home_path = getenv("HOME");
  if (home_path == NULL)
  {
    fprintf (stderr,
     "HOME not found in environment\n");
    exit (HOME_ERROR);
  }

  /*
   * allocate a buffer of the right size to
   * hold both home_path and SNAPSHOT_FILE.
   * The extra 2 bytes are for the '/'
   * separator and the terminating '\0'
   */
  fullpath = malloc (strlen (home_path)
       + strlen (SNAPSHOT_FILE)
       + 2);
  if (fullpath == NULL)
  {
    fprintf(stderr,
      "malloc(): out of memory\n");
    exit (MEM_ERROR);
  }

  /*
   * we could use strcpy() and strcat()
   * here, but since we know that fullpath
   * is large enough, sprintf() is safe.
   */
  sprintf (fullpath, "%s/%s",
    home_path, SNAPSHOT_FILE);

  readfile = fopen (fullpath, "r");
  if (readfile == NULL)
  {
    fprintf (stderr,
      "File '%s': not found\n", fullpath);
    exit (FOPEN_ERROR);
  }

  f = fopen (LOG_FILE, "a+");  /* append? */
  if (f == NULL)
  {
    fprintf (stderr,
      "File '%s': not found\n", LOG_FILE);
    fclose (readfile);
    exit (FOPEN_ERROR);
  }
  printf ("Searching: %s\n", fullpath);

  /*
   * This code makes the same assumption as
   * the original in that input lines are
   * assumed to fit into the 256 byte
   * buffer. Use fgets() as the loop
   * control.
   */
  while (fgets (buffer, sizeof(buffer),
    readfile) != NULL)
  {

    /*
     * Use the same case-sensitive search as
     * in the original code
     */
    if (strstr(buffer, "<title>") != NULL)
    {
      printf("Extracting title\n");
      fprintf(f, "Title: %s", buffer);
    }
    else if (strstr(buffer, "<h1>") != NULL)
    {
      printf("Extracting heading\n");
      fprintf(f, "Heading: %s", buffer);
    }

    /*check for and exit if file write error*/
    if ((rc = ferror(f)) != 0)
      break;
  }

  /* tidy up before returning */
  fclose(f);
  fclose(readfile);
  free (fullpath);

  return rc;
}
```

Balog Pal <pasa@lib.hu>

A fast scan of the code triggers many review comments (which are not necessarily problems, but things to look at more deeply):

- C90-style variable declarations up front
- use of a fixed size buffer
- magic number as size of buffer
- result of `getenv()` not checked
- result from `getenv()` captured as modifiable string ...
- ... which is then actually modified
- `strncat` gets a divine length parameter
- I'm sure could not tell the expected result of re-query "HOME" at the `printf` point
- more magic numbers in code + suspect the previous one could have a typo
- opening file without checking the result
- using `FILE*`s that can be `NULL`
- creating file with hardcoded name at a hardcoded dir that is likely shared
- `fgets` passed limit not related to the passed object
- content of buffer passed to operations expecting 0-terminated string

Can any of those result in crash? Yeah, several of them on proper conditions, there will be `NULL` access if HOME is not in the environment or a file could not be opened; if the file has long lines, there will be a buffer

overrun. And up front I doubt the system allows unrestricted modification of the environment in place.

Let's see the first fuzzy point first, the business on modifying the env. Googling 'unix man getenv' finds the page in first hit. The signature says it returns `char *` (not `const char *`), and the text is silent about modification. Though it states that returned value is `NULL`. Googling 'getenv return string modify' finds what we were about, to a CERT rule [1] entitled 'ENV30-C. Do not modify the string returned by getenv()' with good explanation why. And quoting the C standard that forbids the modification. (I leave the trivia to others that explains the return type and the fact this ruling text is left out of man pages – leaving the world vulnerable to this kind of mistakes.)

So this alone is enough reason to crash the program, no matter if you use `strcat` or `strncat`. In a general situation, `strcat` is a blacklisted function with strong suggestion to use `strncat` or other safe alternatives. So the idea of that fix was good. But not the implementation. For the 'fixed' `str-` functions the size parameter must get the size of the buffer. They prevent the overrun by limiting the range of operation within that length. So they work if you pass them the length of the buffer that is passed as the 'target'. The code passes home-path and 255-1. Those do not relate. You should have passed the length of buffer under home-path, not some random value. Modification this way puts the program in even more danger – as the original is likely flagged in warnings, while the latter is hard to discover.

And if the problem was buffer overrun, in theory fixable by replacing `strcat` to `strncat`, it would still not fix your program, just remove the overrun. Suppose you did copy the result of `getenv()` to a local buffer, and tried to add the filename part there, using the correct length to `strncat:` the result of that is no longer an overrun, but a truncated string that you will use in the next operations without discovering the problem. If the following code got corrected to handle the 'file not found' cases, probably the execution ends there most of the time, but another possibility is it could find a file that has the truncated name. And will process that, producing the result. Sounds like the surgeon executed someone else's operation on you...

Usually I create a fixed version of the code sample, this time I will not, as I could not figure out the intent and there was no specification. Suppose it opened the intended input and output files – after that it is like reading a HTML file, but instead of parsing tags, it tries to read until newline, without provision of long lines, and passes the line as heading or title by finding a tag in it. AFAIK newlines can be spaced quite liberally in tagged files, so this approach will hardly provide anything useful, even if there were no arbitrary length limit. And with the limit it can completely skip discovering the tags if they happen to be split.

The man page for `fgets` states it does produce a 0-terminated string, so if passed an actually correct buffer length – my practice is to use `sizeof(***)` as parameter – `strstr` can be used with defined result. How defined meets expected is another question.

Normally all operations of file must have a check after it to see it succeeded. Failure to open a file is common, it may sit there with the wrong permissions or being locked, the path may be wrong, and many other reasons, it is rude to dump core on those occasions. Write and close may fail due to disk full, a good program not leave a partial/incorrect file without any indication for the problem. All that makes code written in C look like a messy jungle, commands turned to if()-s and 80% of code handling errors. And correctly release all the resources on all the paths asks for discipline and luck humans rarely have in practice.

So the better way is to use C++ and some wrapper class dealing with files that use sticky bit or exceptions to flag errors. That keeps the code readable, while errors can be dealt with at a separate place, also RAII helps to avoid resource leaks. In C++ certainly the problem at the front becomes trivial too, no need for calculating, asking memory, copying, just use `std::string` (or any other string implementation of a used library) and `+` the filename part.

And if the problem was really tied to html I'm sure there are libraries available that parse the whole file, verify it is correct, and allow locating the sections, giving the proper content.

Note

[1] https://www.securecoding.cert.org/confluence/display/seccode/ENV30-C.+Do+not+modify+the+string+returned+by+getenv%28%29

Graham Patterson <grahamp@berkeley.edu>

We are not provided with much in the way of a supporting narrative with this short program. All we are told is that a) it is not working, and that b) the addition of `strncat()` did not help.

Taking a first pass through, it seems that the intent is clear. The program is to read an HTML file and extract lines matching some criteria to the console and a file. There are some interesting issues with file location and overall design. It may be that the author is new to programming as well as C. The first step is to rework the program enough to have it working as intended but without divorcing it from the original. Then we can move on to look at the limitations and possibilities for enhancement. As written, the program appears to be intended for a Unix-like environment, based on the path separators and the use of the HOME environment variable.

The variable declarations at the start of the `main()` function raise one issue at first perusal. The declaration of buffer with a numeric dimension is a prime source for error. This is better done with a symbolic definition, which means either an old-fashioned `#define` or defining a `const int` depending on the antiquity of the compiler.

The next item in the first pass over the code is the assignment of the `getenv()` return to the `home_path` pointer. The string pointed to by the return value of the `getenv()` function is not modifiable by design. The `home_path` pointer is not declared const in any way, so this may be an issue as we continue.

The call to `strncat()` is going to cause problems. This function appends the second argument to the tail of the first argument up to a total length of the third argument. The first argument is a pointer to a string that we have already seen is not intended to be modified. The length argument is another number. Since it is given as '255 - 1', it implies a storage location of 255 characters less one for the trailing null. We do not have control of the length of `home_path`, so this is a problem area. From the limited narrative provided to accompany the code, it may be that `strncat()` was edited in, but the variable usage was not reviewed to conform. There is some support for this conjecture looking at the following `printf()` call, where the format string does not use the `fullfile_directory` or numeric parameters.

The main work of the program is done in a loop which processes the source file one line at a time, and printing lines that match certain criteria to the console and a file. The implementation has a couple of major problems which need to be addressed. The `feof()` call returns the end of file status. The `fopen()` call may not set this even if the file opened is of zero length. So the loop could traverse once on an empty file.

Looking deeper into the loop we see a test for the successful opening of the file. This is misplaced, and should follow the `fopen()` call. Then we have the opening of the output file. A quick scan forward shows that this file is not closed inside the loop, so it will be reopened for every line in the source file. On many systems this will exhaust the number of file handles available to the program. Plus it is a lot of work for no benefit. Add to that, the file is opened for append at end of file (a+). This could be disguised by the repeated `fopen()` call, but will have consequences if the program operates as intended. Then we have the first actual read from the source file, which may trigger the end of file condition.

This loop needs redesigning along the lines of Figure 1.

This scheme ensures that we only proceed with the next step if the files are opened. The file close calls are at the correct level to be called only if the file was opened successfully. Finally, by priming the input buffer before commencing the main loop we ensure that an immediate end of file

Listing 2

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

int main()
{
  const int buffersize = 255;
  const int inputlimit = 254;
  const char * srcfile = "/snapshot.html";
  char home_path[buffersize];
  char * fullpath = NULL;
  char fullfile_directory[buffersize];
  char buffer[buffersize];
  char *outputfile = "/tmp/output.log";
  FILE *f;
  FILE *readfile;
  if(strlen(getenv("HOME"))
     < inputlimit - strlen(srcfile))
  {
    strcpy(fullfile_directory,
     getenv("HOME"));
  }
  else
  {
    exit(1);
  }
  strncat(fullfile_directory,
    srcfile, inputlimit);
  printf("Searching: %s\n",
    fullfile_directory);
  readfile = fopen(fullfile_directory, "r");

  if(readfile)
  {
    f = fopen(outputfile, "a+");
    if(f)
    {
      fgets(buffer, inputlimit, readfile);
      while (!feof(readfile))
      {
        if (strstr(buffer, "<title>"))
        {
          printf("Extracting title\n");
          fprintf(f, "title: %s", buffer);
        }
        else if(strstr(buffer, "<h1>"))
        {
          printf("Extracting heading\n");
          fprintf(f, "Heading: %s", buffer);
        }
        fgets(buffer, inputlimit, readfile);
      }
      fclose(f);
    }
    else
    {
      fprintf(stderr,
        "Unable to open output file '%s',\n",
        outputfile);
    }
    fclose(readfile);
  }
  else
  {
    fprintf(stderr,
      "Unable to open source file '%s'.\n",
      fullfile_directory);
  }
  return 0;
}
```

Figure 1

```
Open the source file
If the open is successful,
  Open the output file
  If the open is successful,
    Read the first source line
    While not end of source file
      Test the line and output as required
      Read the next source line
    End while
    Close the output file
  End if
  Close the source file
End if
```

condition is handled correctly. Adding else clauses gives us some error handling options.

A revised version is provided as Listing 2. This uses a pair of **`const int`** declarations to abstract the hard-coded buffer sizes, which lead to a probable typographical error (255 versus 225) and a possible overflow for **`buffer[]`**. The fixed input and output file names have also been abstracted to variables. All sizes have been corrected to use the variables. The string returned by the **`getenv()`** call has been stored. I have also included a simple check to confirm that the intended buffer is large enough. The HOME environment variable does not typically return a long string, but testing the reality costs little in effort. The file read loop follows the pseudo-code outline already given, with basic error reporting.

The astute reader will have spotted several issues with the design. Without a supporting narrative we can only comment. Hard coding filenames and paths is uncommon. Using different files or locations would require an edit and recompilation. On the other hand, unless command line arguments are used, or some form of configuration file employed, this is not unreasonable for a proof of concept piece. Having the source in the user's home directory, but the output in `system /tmp` is unusual. This is usually system scratch space and is often flushed on a reboot.

Opening the file for append at end means that repeated execution will extend the output file. A simple open for write might be a better fit to the original author's intentions. The **`strncat()`** function has one quirk that would have affected the original. It '\0' pads the destination to the length requested. [Ed: this is the behaviour of **`strncpy`**.] In the original code this would have exceeded the allocated space by a significant margin.

The act of opening a file does not necessarily set the end of file state, even if it is of zero length. A read at end of file will, so the source file should be read before entering the while loop. The original code could issue multiple **`fopen()`** calls on the same file, and also issued **`fclose()`** at the end irrespective of a successful open. Depending on the environment, this could cause exhaustion of available file handles. The **`fclose()`** calls are misleading as to the actual scope of the open files. No use is made of the return from the **`fgets()`** or **`fprint()`** calls to check for errors on the streams. The end of file occurrence is already covered, and it may be argued that the issue of a read or write error is too much for a small program.

Finally, the design does not make allowance for the real-world format of HTML files. This design will not generalize. The tags detected are only opening ones, and no removal of leading content is performed. Unless the HTML source is structured one tag pair per line (which is possible, of course), the output will not be clean or complete. Handling this eventuality would require a more sophisticated design, and would definitely exceed the scope of the original.

All in all, a lot in less than 50 lines. The suggested alternative code was developed and tested using Xcode on OS X 10.4 for a Unix-like environment.

Joe Wood <joew@aleph.org.uk>

Derek sat down and Brian pushed a pint in his direction and said, 'You're going to need that.'

Sounding more grumpy than he intended Derek said, 'Joe?'

Brian smiled and said, 'Who else comes to us with their programming assignments?'

Joe was well known to the two post graduate students. He could be a pain but was enthusiastic and keen to learn, and they both knew that everybody had to start at the beginning. Besides Dr. Möbius had made it clear that helping their fellow students would improve their skills, exactly how or why was, in their opinion, less obvious.

'OK,' said Derek. 'You had better show me the problem.'

Brian had already turned on his laptop, and said, 'Well, it's really quite simple. The program opens a hard-coded HTML file in the user's home directory, and prints out any title or top level headings elements that it finds therein.'

'That's OK, what about HTML structure and semantic rules?' asked Derek, wondering if this was going to turn into an HTML parser.

'Slow down, it's just a straight line-by-line text search and display program. We find the required start tag (**<title>** or **<h1>**) in the line (if any) and display the line,' Brian said calmly, trying to keep Derek from going off on some over-engineered solution.

Derek said haltingly, 'No semantic processing, no checking for invalid HTML, no looking for nested elements, no looking for the end of the required elements?'

'Absolutely not. It's a simple text searching problem. But we do have to record the found lines in a log file in `/tmp`.' said Brian.

'We can do that!' said Derek warming to the new task. 'So what exactly is Joe's problem?'

Brian showed him Joe's code, and said, 'Joe came to me because it keeps crashing when he concatenates the two parts of the file name. Joe said he originally tried **strcat** and replaced it with **strncat**. But the program still crashes in the same place.'

Derek looked at the offending code and said, 'Well of course it will crash, **home_path** is a string pointer returned by **getenv**, and it points into the environment list and must not be mucked about with.'

'I remember, the environment list contains a table of name=value pairs, so Joe is trying to write into what is effectively the OS's private data.'

'Exactly.' said Derek. 'Now, if we replace the offending lines with'

```
home_path = getenv ( "HOME" );
fullfile_directory =
  malloc ( strlen(home_path) +
           strlen( "/snapshot.html") + 1 );
strcpy ( fullfile_directory, home_path );
strcat( fullfile_directory, "/snapshot.html");
```

'Then, we just copy the value returned by **getenv**, append the required file and we can do what we like to that,' said Derek.

'Hold it, there are two problems,' said Brian. 'Firstly, **getenv** may fail, especially if we made a typo in its argument. Secondly, and more generally, what about error handling? We really should check the results of functions like **getenv** and **strcat**.'

'Right, we should test the values returned by library functions. However, this is a student's project and it is far from clear what we should do if a function fails.'

'OK,' said Brain, 'let's just call **error**. I know, it is a gnu extension, but it does provide a simple straightforward method for reporting errors and quitting. So for example we can use.'

```
if (!home_path) {
    error ( EXIT_FAILURE, errno,
            "Unable to determine home path" );
}
fullfile_directory =
    malloc ( strlen(home_path) +
             strlen( "/snapshot.html") + 1 );
```

Derek said, 'We can do better than that. If we encapsulate the call to **error** in a local procedure, **test_status**, and pass the condition to be tested into that, then if we report an error it will just never return, like so.'

```
static void test_status( bool err_result,
  int errnum,
  const char *format, ...)
{
  if ( err_result ) {
    va_list args;
    va_start ( args, format );
    error(EXIT_FAILURE, errnum, format, args);
    va_end ( args );
  }
}
```

Brian though for a few moments and said, 'In any case, there is subtle issue with using **strncat**. **strncat** offers the safety value of only copying a specified maximum number of characters, which can be useful, if the source has no terminating null. The terminating null is then always added. However, there is a catch. If the supplied length is larger than the supplied source string, then the length is made up by appending extra nulls. This could contribute a significant time penalty.'

Derek said, 'We never use fullpath. So the first part of the solution becomes.'

```
static char * cat_path ( const char * part_1,
                         const char * part_2)
{
  // Note: Although this procedure looks
  // lengthy and complicated, in fact because
  // of its use of memcpy, the compiler
  // produces some efficient code.

  // Get the lengths of the 2 part strings
  const size_t len_1 = strlen(part_1);
  const size_t len_2 = strlen(part_2);

  // Allocate space for full file name
  char * full_name = malloc ( len_1 + len_2 + 1 );

  // Check that allocation was ok
  test_status( !full_name, errno,
    "Cannot allocate memory for file name "
    "in cat_path");

  // Copy part_1 name into full_name
  memcpy ( full_name, part_1, len_1 );

  // Append part_2

  // Note: the additional byte is copied
  //       to pick up the trailing null
  //       from the part_2 string on the fly
  memcpy ( full_name+len_1, part_2, len_2+1 );

  return full_name;
}

int main ()
{
  const char * home_path = getenv ( "HOME" );
  test_status( !home_path, errno,
    "Unable to determine home path" );
```

```
  const char * fullfile_directory =
    cat_path ( home_path, "/snapshot.html" );

  // The printf should just have only two
  // arguments
  printf("Searching: %s\n",
    fullfile_directory);

  // File handle for output log file
  FILE * f;

  // File handle for reading from
  // fullfile_directory
  FILE * readfile;
```

[To be continued …]

Brian said, 'How what about the file opening?'

Derek replied, 'Well Joe even managed to get that wrong. First off he never tested the return statuses. Secondly, the second argument to open the output file only needs to be '**a**' and not '**a+**'. Thirdly, and most importantly, he opens multiple output files, one each time around the feof loop, which causes the output to be written in reverse order.'

Brian smiled, 'That's quite impressive for two **fopens**. A better solution would be.'

```
  readfile = fopen(fullfile_directory, "r");

  // Check input file was opened ok
  test_status( !readfile, errno,
    "Cannot open file %s for reading",
    fullfile_directory );

  // Just append to the file
  f = fopen ( "/tmp/output.log", "a" );

  // Check output log was opened ok
  test_status( !f, errno,
    "Cannot open file %s for writing",
    "tmp/output.log");
```

Brian said, 'I would put both into small local procedures so that I need not repeat the file name details, but it's more a matter of personal style.'

Derek said, 'And finally we get to the real business of reading the HTML file. Something like.'

```
  char buffer [BUFFER_SIZE];

  // Main processing loop
  while ( read_line( buffer, BUFFER_SIZE,
            readfile, fullfile_directory ) ) {
    look_for_token ( buffer, "<title>",
      "Title", f);
    look_for_token ( buffer, "<h1>",
      "Heading", f);
  }

  // Close opened files etc.
  fclose(f);
  fclose(readfile);

  // And exit normally
  return 0;
}
```

'That tidies up several errors with the size of the input buffer. **buffer_size** is a macro, say 256 bytes.'

Brian said, 'Now, **look_for_token** is quite straight forward.'

```
static void look_for_token (
  const char * buffer,
  const char * token,
  const char * message,
  FILE * f)
{
    if ( strcasestr ( buffer, token ) ) {
      printf ( "Extracting %s: \'%s\'\n",
        message, buffer);
      fprintf ( f, "%s: %s\n",
        message, buffer);
  }
}
```

Derek said, 'You know that **strcasestr** is another gnu extension, but its quite easy to replicate. It's just a case insensitive search. You do realise that you have slightly changed the output text?'

'Yes, but it does allow for future expansion.'

Brian said 'Finally for **read_line**.'

```
static const char * read_line (
  char * buffer,
  size_t size,
  FILE * stream,
  const char * name )
{
  const char * input =
    fgets ( buffer, size, stream );

  if ( input == NULL ) {

    // fgets returns NULL both on error and
    // eof, so we must test for eof before
    // assuming the worst
    if ( !feof(stream) ) {
      error(EXIT_FAILURE, errno,
          "Cannot read data from %s", name);
    }
  }

  return input;
}
```

Brian said, yawning, 'Well, better test it with some sample HTML.'

'Of course. Here's some I prepared earlier.'

A few tests later. 'Good. I think Joe owes us another round. What about Dr. Möbius?'

Commentary

As most of the entries said, the code under critique is quite badly broken and also the exact purpose of the program is non-intuitive.

My particular interest here is with **strncat**, which is a rather odd function and curiously rather different from **strncpy**. Here is an extract from the specification for this function in the C standard:

The **strncat** function appends not more than n characters (a null character and characters that follow it are not appended) from the array pointed to by **s2** to the end of the string pointed to by **s1**. The initial character of **s2** overwrites the null character at the end of **s1**. A terminating null character is always appended to the result. Thus, the maximum number of characters that can end up in the array pointed to by **s1** is **strlen(s1)+n+1**.

In my opinion this is a poor design. One easy assumption to make is that the last parameter is the length of the target buffer – this matches the use of the argument in **strncpy** and some other similar functions. Even avoiding that error the user of **strncat** must calculate the amount of space left in the target string, remembering to allow for the null terminator.

Listing 3

```
#include <stdio.h>
#include <iostream>
#include <stack>
using namespace std;
stack<char> cStack;

void decToHex(int num){
  int showNum = num;
  int storeNum;

  while(num != 0){
    storeNum = num % 16;
    switch(storeNum){
      case 10:
        cStack.push('A');
        break;
      case 11:
        cStack.push('B');
        break;
      case 12:
        cStack.push('C');
        break;
      case 13:
        cStack.push('D');
        break;
      case 14:
        cStack.push('E');
        break;
      case 15:
        cStack.push('F');
        break;
      default:
        cStack.push( (char)storeNum );
        break;
    }
    num = num / 16;
  }
  cout << showNum << " in hexadecimal is ";
  while(!cStack.empty()){
    cout << cStack.top();
    cStack.pop();
  }
}
int main( int argc, char ** argv )
{
  // test it
  int integer;
  cin >> integer;
  decToHex( integer );
}
```

It is easy to get this wrong, and then **strncat** loses its buffer overrun protection.

When combined with the behaviour of **strncpy** (which does not append a null terminator when the source string is longer than the target buffer) it is all too easy to overflow the target buffer.

The design also breaks a symmetry between **strcpy** and **strcat**; if the destination buffer is empty, **strcpy(dest, src)** and **strcat(dest, src)** have the same effect, but **strncpy(dest, src, n)** and **strncat(dest, src, n)** have different effects.

Sticking to C makes it hard to avoid these errors; in C++ use of the standard string class does all the buffer allocation behind the scenes.

The Winner of CC 58

It was very hard to choose a winner from the entrants this time – the critiques covered most of the issues fairly comprehensively. There was some disagreement about the precise semantics of **strncat** – which was one of the points raised by the critique – and I don't think anyone got it right!

So the solutions deciding to allocate memory dynamically are probably the safest way to go – although it is important to remember to free the resultant string.

Overall I felt that Pete Disdale's critique was the best by a short head, and so he will receive the one-off prize for this issue of the Code Critique: a family ticket to visit Bletchley Park.

Code Critique 59

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

I'm trying to write a decimal to hex converter but it doesn't quite work – can you help fix my code.

While you're there, there a number of additional comments you might wish to make.

The code listing is shown in Listing 3. You can also get the current problem from the accu-general mail list (next entry is posted around the last issue's deadline) or from the ACCU website (http://www.accu.org/journals/). This particularly helps overseas members who typically get the magazine much later than members in the UK and Europe.

Inspirational (P)articles

Frances Buontempo introduces Linda Rising's inspiration.

In CVu May 2009, I shared the inspiration created by reading an interview with Donald Knuth. I want this to become a regular feature because I passionately believe sharing positive experiences causes ripples, as though the inspiration spark permeates through the ether and hits surrounding people. Terry Pratchett has written about such inspiration particles in various books, hence the title of this series.

Many thanks to Linda Rising for sharing this encouragement to step away from the keyboard for moment.

If you have a story to share of a recent uplifting or encouraging moment, please send it to frances.buontempo@gmail.com.

Agility at a personal level: Implications for daily life choices

Our favourite drug (way ahead of nicotine and alcohol) is caffeine. It's the first thing we reach for in the morning. It allows us to be our perky best and fell energized. But does it represent the best, the most agile way to navigate our day?

Seredipity brought us coffee and tea just as factories appeared, heralding the Industrial Revolution. Almost overnight, instead of waking with the sun, we work to a SCHEDULE. We had to be on time and stay that way throughout the working day, sleeping less and less over the years. Some of this was good, of course. The lives of ordinary people improved. Boiling water meant producing a safer drink.

Is it time to get off the treadmill we boarded at the start of the Industrial Age? Is it time to question the myth that we are the best problem solvers when we force it by working without stopping until we have readed our goal (or we fall asleep over the keyboard)? How about some experiments where some of us work in short cycles with breaks and enough time for sleep and other interests? Sounds quite agile! I believe we might discover the our creativity, productivity, and happiness soar!

ACCU Security – Yesterday, Today, and Tomorrow

accu security
yesterday, today, and tomorrow

ACCU announce a 1-day conference at Bletchley Park.

On November 7th 2009, ACCU will be holding a one day conference at Bletchley Park, home of the legendary World War II 'Engima' code breakers, and the site at which the world's first digital computer went operational.

Confirmed speakers, in alphabetical order, include:

- Tony Sale, lead on the working reconstruction of 'Colossus', the world's first digital computer. The original was used to break the German 'Lorenz' code, and played a vital role in the run up to the Allied invasion of Europe.
- Simon Singh, author, journalist and TV producer, specialising in science and mathematics, and the author of *The Code Book*, a history of codes and code breaking from Ancient Egypt to the Internet.
- Phil Zimmerman, the original creator of the PGP email encryption package, which despite three years of government persecution became the most widely used email encryption software in the world.

The Conference will be held in the elegant Victorian Bletchley Park Mansion, at the centre of Bletchley Park itself, allowing conference attendees the opportunity to visit the exhibits on show at the National Museum of Computing and the rest of Bletchley Park. Bletchley Park is home to a number of unique artifacts, including the Colossus, the Bombe (including the mock-up that featured in the film 'Enigma'), original Enigma machines, and a Lorenz coding machine.

The proceeds of the conference will go to the Bletchley Park Trust to help with the upkeep of Bletchley Park. Conference rates have not, at time of writing, been finalised but will be in the region of £95 per person.

More information and registration details are on the website at: http://accu.org/index.php/conferences/accu_conference_2009_security.

Write for us!

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

What do you have to contribute?

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

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

Bookcase

The latest roundup of book reviews.

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

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

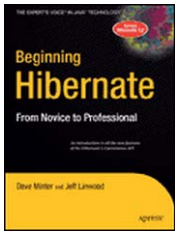
I must thank Blackwells and Computer Bookshop for their continued support in providing us with books.

Jez Higgins (jez@jezuk.co.uk)

Beginning Hibernate – From Beginner To Professional

By Dave Minter and Jeff Linwood, published by Apress, ISBN: 978-1-59059-693-7, 317 pages

Reviewed by Andrew Marlow



Recommended with reservations.

The contents appears at first glance to promise coverage that will take the reader from novice to professional. It starts with an introduction (mentioning the purpose behind ORM tools), configuring Hibernate, a simple application, persistence lifecyle, mapping (both via mapping files and via annotations), sessions and searches. There are a number of appendices covering advanced features (e.g. hand-rolled SQL, invoking stored procedures), tools (Ant and Eclipse), Spring, and upgrading from Hibernate 2.

Although the table of contents looks promising, appearances can be deceptive. The book does not take the reader from novice to professional in the way you would think. Instead of an initial shallow overview that covers the essential topics at a novice level, followed by covering the topics in greater depth in later chapters, each topic is covered completely before it moves onto the next one. This does mean everything is covered, and in enough depth to convey the level of professional knowledge it promises. However, it is very difficult for the novice. For example, the first complete working program simply saves a POJO. That program is not built on later, as you might expect. The reader does not see how a table is queried until chapter 9 (searches and queries) which is two thirds of the way into the book. This accounts for my recommendation but with reservations.

The failure to cover all the fundamental relational database tasks (create, read, update, delete, transactions) at a novice level, and then refine them in further chapters, is quite a failing. It is not necessary for every book to take this approach but it is particularly useful for a database book to do so. You rarely perform just one of these tasks in isolation. Hence, you would expect it of a book that is subtitled 'from novice to professional'. This flawed approach shows in several ways. For example, it is a long time before enough knowledge is built up for the reader to be able to create a realistic Hibernate program at all. The examples focus on the micro-detail leaving the reader in the dark for the bigger picture. Also such detail can be overwhelming for the beginner.

The division between the main text and the appendices seems a bit strange. Why bother having an appendix on moving from Hibernate 2 to 3? Anyone who wants to move from 2 to 3 will already have most of the knowledge covered by the book and they would not buy the book just for this appendix. Also, why relegate Hibernate and Spring to an appendix? These are often used together. Generating the schema from the mapping files get insufficient coverage in the main text. The reader is only shown how to run the generation from Ant in an appendix.

There is lots of example code, but the detail obscures how Hibernate would be used at a strategic level in a project. For example, transactions are covered (in the sessions chapter) but there are no examples of how a business level operation that involves more than one table would have to use a transaction to ensure consistency (the @transacational annotation is not covered either).

Having said all that, the book does contain alot of useful information, and each idea is presented in a reasonably clear and helpful way. Also, it is succinct and avoids the distracting chatty style that seems to be so common these days. I found this book better than other Hibernate books I have seen. However, a novice would probably have to read the book twice to get the best from it. This is because of they way it covers each topic completely before moving to the next one and also due to the succinct presentation.

I am waiting to see a better Hibernate book than this one, but for the time being, this will have to suffice as the best one I've seen. It would be even better if it had a more accurate title, it is not really suitable for beginners.

Bookshops

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

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

View From The Chair

Jez Higgins
chair@accu.org

It may surprise you to know that, despite my position as Chair, ACCU is not the only organisation of which I am a member. I am also a paid up member of a cyclists organisation, Audax UK. AUK membership is open to any cyclist, regardless of club or other affiliation, who is 'imbued with the spirit of long-distance cycling'. They act as an umbrella for events organised by members, run a mailing list, and publish a magazine. I have discussed lifting a feature from the Audax magazine with Steve. Each issue has many pages of photos, but all the photos are of AUKers, many of whom have beards, riding their bikes. We could fill CVu with photos of ACCUers, many of whom have beards, at their desks.

'Imbued with the spirit' is a fantastic phrase. You don't actually have to do any long-distance cycling, you have to think you might at some indeterminate point. That had, indeed, been my position for some time until, on the 21st of June, I rode my first audax.

An audax is a long-distance bike ride. You're given a route card, with directions, and there are a number of control points along the way. At the control points you stop and get your card stamped and your arrival time is recorded. The controls are generally at cafes or pubs, so you can stop and have a drink and a piece of cake too.

My day went something like this...

Arrive in good time for the 8 o'clock start. Many people already there, greeting each other with 'I haven't seen you since ...', 'How did you get on at ...', and so on. I don't know anyone, although I think I recognise a couple of names. Eat a banana.

Everyone crowds to the start. We're off. Ride for an hour and half to the first control. Have a coffee, and fall into conversation with another chap. We ride together to the next control. We talk about bikes. I eat a banana.

At the next control, we fall into conversation with some other chaps. We talk about bikes. We ride out together to the third control. On the way, I eat a banana.

Last control and everyone's starting to flag a bit. I have to be guided through ordering a sandwich. We talk about the ride so far, and what's left to do. On the last stretch, I fall in with a couple of chaps who, it turns out, live just round the corner from me. Eat another banana.

204 kilometres and 10 hours or so later, I make the finish (or arrivee in audax speak). I'm hungry and tired. Not physically tired, particularly, but mentally. My new ride-chums are asking me if I've enjoyed myself, but I can no longer form proper sentences. I head home, determined to do it again.

It takes a little bit of imagination (and a few less bananas), but my first experience at an ACCU conference – JaCC 2000 – was very similar. Arriving in a place full of people who already seem to know each other, long periods of concentration, interspersed with coffee and biscuits, talking to people about what's just go on and what's coming up, leaving completely exhausted, but having had a bloody marvellous time regardless of how ridiculous that might have seemed at the beginning.

Membership

Mick Brooks
accumembership@accu.org

You're reading this just after the traditional membership renewal period, August, is over. I'm writing it just as that period begins, so I can't tell you how it went. If you're an August member and are reading this, then you managed to renew without problems. If you didn't, then you won't be reading this, so giving instructions on how to renew seems somewhat futile. However, it can't hurt, especially since newer members will have their renewal come up at different times of the year.

The preferred way to renew is by logging in to the website, then following the links named 'Account', then 'ACCU Subscriptions'', and then 'Renew', where you can pay by credit or debit card. I'll happily accept a cheque if you'd prefer (email me for details). Some of you will have arranged to pay by standing order, and won't have to do anything to ensure your membership continues. If you're not currently paying by standing order drop me a line to find out how to set one up. Less hassle for you, and we even give you a discount.

In any case, now is a good time to log in to the website and review your mailing address details and contact preferences. If you have problems or questions about renewals, or anything else, then email me at accumembership@accu.org.

Advertising

Seb Rose
ads@accu.org

This has been a bad year for advertising in general and ACCU advertising in particular. There has been a marked decline in new enquiries and some of our existing advertisers have decided not to renew. So... please alert your employers, suppliers and customers to the possibility of advertising on the ACCU website and in the journals.

With regret we announce the death of Adrian Leigh Gothard, who died on August 6th following a long battle against illness. During the 1990s, Adrian was a regular contributor to CVu and spoke at the ACCU Forum on the subject of embedded systems.