

Contents

Reports & Opinions

Reports

Editorial	4
From the Chair, Secretary's Report, Membership Report, Advertising, Standards Report	5

Dialogue

Student Code Critique (competition) entries for no 22 and code for no 23	6
Letters to the Editor	9
Francis' Scribbles	10

Features

On Not Being a Software Engineer <i>by Simon Sebright</i>	12
ACCU Spring Conference 2003 Roundup <i>by Thaddaeus Frogley</i>	13
Professionalism in Programming #21 <i>by Pete Goodliffe</i>	18
Maintaining Context for Exceptions <i>by Rob Hughes</i>	21
When Worlds Collide #1 - Embedded Systems and General Purpose Computers <i>by Mark Easterbrook</i>	24
Mixing Strings in C++ <i>by Sven Rosvall</i>	25
A Polygon Seed Fill Algorithm <i>by James Holland</i>	27

Reviews

Bookcase	29
----------	----

Copy Dates

C Vu 15.5: September 7th

C Vu 15.6: November 7th

Contact Information:

Editorial: James Dennett
914 24th Street,
San Diego
CA 92102, USA
cvu@accu.org

Advertising: Pete Goodliffe
Chris Lowe
ads@accu.org

Treasurer: Stewart Brodie
29 Campkin Road,
Cambridge, CB4 2NL
treasurer@accu.org

ACCU Chair: Ewan Milne
0117 942 7746
chair@accu.org

Secretary: Alan Bellingham
01763 248259
secretary@accu.org

Membership Secretary: David Hodge
01424 219 807
membership@accu.org

Cover Art: Alan Lenton
Repro: Parchment (Oxford) Ltd
Print: Parchment (Oxford) Ltd
Distribution: Able Types (Oxford) Ltd

Membership fees and how to join:

Basic (C Vu only): £15
Full (C Vu and Overload): £25
Corporate: £80
Students: half normal rate
ISDF fee (optional) to support Standards work: £21
There are 6 issues of each journal produced every year.
Join on the web at www.accu.org with a debit/credit card, T/Polo shirts available.
Want to use cheque and post - email membership@accu.org for an application form.
Any questions - just email membership@accu.org

Reports & Opinions

Editorial

Next Steps

The only thing that is guaranteed in the programming world is change. We hope that it is progress, but that's more controversial. As the programming world changes, so we too change, updating our skills, specializing or diversifying as we choose, learning new things. Being the sum of its members, it is natural that ACCU also changes. It has grown and morphed significantly in the last few years, and will continue to do so.

The last two years have been something of a transitional period for ACCU: two major players have, at least partially, stepped aside in the last year or two to make way for new blood. Even so, ACCU still grew last year, and is well positioned to continue to grow. (As an aside: maybe we should not take it for granted that growth is a good thing, but it seems to me that it is an inescapable side-effect of being good at what we do. If ACCU is valuable to existing members, it will be good for others and so will naturally tend to grow.) That growth will be fuelled by the actions of members, whether in filling existing roles or suggesting new paths to tread.

It has taken me since I took over the editorship of C Vu at the start of 2002 to feel that the handover period from the previous editor is really at an end. There have been glitches en route, certainly – thanks go to those who have politely pointed them out – but with help from the outgoing editor Francis and from our remarkably patient production editor Pippa none has been disastrous. The production editor's role, incidentally, covers just about everything about the journal except the contents of the articles. If you notice that C Vu (and Overload) look much more professional than the output of most user groups, it is the production editor to whom the credit should go.

Now that C Vu is through this transition, it is time to think about its future. There are many ideas on how C Vu can be made better – quicker/better review of articles, better suggestions for new articles, better use of the ACCU website to give information about the journal and to publish the source code that sometimes accompanies articles, and who knows what else. These and many others are good ideas, but I cannot do them all alone, and would not get the best results if I tried. I have recently had one offer of some assistance – would anyone else who has opinions on what they'd like from a future version of C Vu care to do something about it? If so, either get in touch with me (addresses on the contents page) or with Tom Hughes, the publications officer on your committee. Roles are flexible, and compensation is in the form of job satisfaction and maybe something to write on a CV (or resume if you're on this side of the pond). One note: please do not assume I'm addressing only some core group of familiar names within ACCU here. New blood is not only permitted but positively encouraged.

Patent Nonsense?

While ACCU's membership includes hobbyists and students as well as those who work in software

development, the majority of those I know are professional software developers of one kind or another. Many of us work in software development companies. (Given the need to refer to everything in the industry by a TLA, these are often referred to as ISVs, for Independent Software Vendors.) Software companies' main asset is rather intangible: it's certainly not computer hardware, as that loses most of its value as soon as it is unpacked and usually most of the rest in the following 24 months, and there's no significant wealth in digital media or user guides. What is valuable is something else, something that is frequently labelled "Intellectual Property", and again this is abbreviated to IP, just to confuse matter in an industry where the acronym IP already had enough meanings.

For now, I'm going to make the (rash) assumption that this "IP" has value even if the creators leave the company. (One more diversion: companies who declare that their employees are their most valuable assets only just miss the point. Their people are free to leave at any time, give or take a notice period. The real asset a company can have is its ability to retain good employees.)

Some years ago, when I was learning martial arts, we had been working one evening on some techniques which lead to sitting astride one's opponent while he was on his back on the floor, with his arms pinned down by knees and legs and one's own arms and hands free to attack. (For the pacifists among you, imagine that I'm describing a particularly beautiful piece of origami. It might or might not help, but it won't do any harm.) Colin, our instructor, then announced that this position was an excellent one from which to start a fight – because you're going to win. I recommend the study of martial arts to any software developer: the combination of physical and mental exercise is an ideal way to leave the troubles of work behind.

It is not a good idea to use the term Intellectual Property without consideration. Advocates of draconian restrictions on how non-tangible goods can be used like to join many of them (copyrighted works, ideas protected by patent, trademarks) together and refer to them as property before the debate over how these goods should be used.

Possibly supporters of this position studied the same ideas Colin taught. If they can make enough people start the debate with the assumption that all intangible goods are property, you're likely to win: it's not going to be too hard to argue by analogy that they should be protected in similar ways to tangible goods. Lawmakers can rarely have the technical understanding to see that treating (say) software as if it were (say) house bricks makes no sense, but they do like to be able to apply existing bodies of law to new situations. Businesses whose interest is in claiming ownership of ideas can and do lobby effectively to get laws backing their position.

Much has been written on why the term IP is a disingenuous one, but yet complete refusal of programmers to cooperate is not an acceptable option within corporations that do rely on selling software to keep in business. The people running these businesses are usually comfortable with the idea of selling property to make a profit, and so it is uncomfortable for them to consider alternatives.

The job of a software developer divides into some coding/"pure" software stuff and judgment calls, some of them non-technical in nature. Many end up being ethical decisions:

- What should we do when we're told to drop quality?
- What should we do when bad decisions lead to pressure to work insane hours?
- Which position should we take in the debate on ownership of ideas?

At work, we're rarely just programmers. We have more or less desire/success to work purely on aspects of projects relating directly to software, but few can survive far into a career without having to spend considerable time doing things that are some way removed from the sharp end of writing code.

Companies are driven by money. There was a brief time, in the dotcom bubble days, when it seemed acceptable for companies to be driven by technology, but those days are gone (and mostly for the best). Those who control a company and its finances do not usually also have the time to be technical experts. That's why they employ us. Sometimes though the boundaries are blurred. A process to "capture Intellectual Property", for example, is likely to be motivated by financial/legal goals, and might well read like something that has been written more by lawyers than by anyone who understands software development. At least once in my career I looked at the existing defined process for capturing "IP" and knew that it could not reflect the reality of what happened in the organisation. Agile organisations and heavyweight protocols in the workplace don't mix. Process improvement works best, in my experience, in small increments – in fact, the ideas of agile methodologies apply just as well to software process and organisations as they do to the artefacts that we produce (code, documentation, test data etc.) On occasion it is necessary to have revolution rather than evolution, but when possible the quickest route between two points is often, seemingly paradoxically, found by taking the greatest possible number of small steps.

Each small step is easier to define and more likely to succeed than a drastic change, and success breeds success. Making some effort to document and communicate the creative works of a software development group is worthwhile for a number of reasons, however. It's something your organisation probably ought to do. If you get there first, with a lightweight process that still gives the business what it needs, you might avoid having a process mandated without such consideration for the realities of software development.

...Now Get The Book

For all the criticism of C++ as suffering from a volatile evolution, the situation of many of us who use C++ compilers has improved significantly since the C++ Standard was officially ratified in 1998. Compilers released in the last year or two are generally fairly close to implementing the standard, and porting code between them has become significantly less burdensome. There were many glitches in the published C++ standard, some of which generated far more heated conversations than they warranted – such as the omission of a guarantee that `std::vector<T>` (for T other

than `bool`) is contiguous so that it can be used to interface with code using raw (C-style) arrays. Five years on, an updated version of the C++ Standard has been published, including fixes for many errors that made it into the original 1998 document. The current C++ standard, as of April 1 2003, is ISO/IEC 14882:2003(E), replacing 14882:1998. The good news is that C++2003 does not introduce new features, and breaks little if any existing code. The other good news is that it will, for the first time, be published in book form later this year.

C last had a major facelift in 1999, and has also published one corrective update in the interim. While adoption of C99 (as the 1999 C standard is informally known) has been slow, it has beaten C++ to publication in book form. To quote Francis Glassborow: "The current C Standard (C99 + TC1 folded in) and the rationale is now available in book form published by Wiley (ISBN 0-470-84573-2). It is a hard cover lay flat binding with almost 800 pages. For once you might find it cheaper to buy in GBP (34.95) rather than US\$ (65)." (TC1, for those who spend their time doing things other than learning minutiae of the ISO standards process, is the first "technical corrigendum" to a standard – in more familiar terms, a service pack for a technical document.) Unlike a previous printing of the 1990 C Standard, which was accompanied by annotations considered by many to be a liability rather than an asset, these books just give the standards documents as they were meant to be read. These aren't books for beginner to intermediate level programmers to use when learning C++, but they are the ultimate references. For those who want something more portable than the \$18 PDF versions, without spending the amazing sums the standards groups charge for a printed copy, this is good news.

James

From the Chair

Ewan Milne <chair@accu.org>

I certainly can't complain about ACCU Chair being a dull job so far, as already the challenges are popping up frequently enough to keep me on my toes. I hope by the time you read this we will have an organiser for next year's conference: I can't, however, confirm further details at the moment.

Recently I went along to the launch of a new exhibit on home computing at the Museum of Computing in Swindon. This is the first museum in Britain dedicated to computing, and opened in February this year. Still a small concern in the process of being fully established, it definitely deserves your support. The enthusiasm and expertise of the curators was infectious as we were taken back to the glory days of the 70s-80s home computing boom – an era in which I imagine many ACCU members first caught the computing bug.

Thanks to the hands-on element of the exhibit, I had great satisfaction in proving I could still knock out a "Hello, world" program on a ZX-Spectrum,

and remembered the schoolboy glee with which I used to torment shop assistants with the Oric-1's 'Zap' and 'Explode' commands. The home computing exhibit runs until October, see www.digitalhistory.org.uk for details. I must point out that my recommendation is in no way connected to the fact that I won the prize draw at the launch, though the prize of an original ZX81 was much appreciated, and now I just need to get my hands on a power station for it to run...

Secretary's Report

Alan Bellingham <secretary@accu.org>
Reports from the Secretary are quite rare - I'm mostly responsible for the administrivia of the committee and, so long as that is running smoothly, there is little for me to say. However, this time I do have an announcement that is relevant to our members.

As those of you who were present at the AGM back in April or who have scrutinised the Officers' section in this year's Handbook will be aware, our previous treasurer, Bryan Scattergood, has stepped down from the role. We were pleased to welcome Paul Johnson as his replacement.

Sadly, circumstances beyond his control mean that Paul was unable to take up the post. Instead, we are glad to accept Stewart Brodie who has stepped in to take his place. Stewart is an existing member of the committee, and we are grateful that his accession should mean the minimum amount of disruption in one of our most important offices.

Naturally, this appointment is provisional on confirmation by the members at large at the next AGM.

Paul also remains a committee member, and will be tackling the previously vacant Publicity post.

Membership Report

David Hodge <membership@accu.org>

The final total for 2003 was 1125, 15 more than last year.

We are now in the renewal cycle and everyone should have received either an email or a normal mailing requesting that they renew their membership. If your renewal can be completed before the end of August it makes this job so much easier. At the time of writing (end of July) 20% of the membership have renewed. The simplest way for me is for you to renew via the website <http://www.accu.org> as I can then process the data automatically.

To those overseas members who responded to the request for data on the receipt times of their journals - thank you. The approximate times for receipt after the UK members have had theirs are Europe 3-7 days, USA 7-17 days (US mail service now ships across USA by road, which adds to the delay), rest of world 6-17 days. There is not a lot we can do to improve this as we are using just about the fastest system for bulk mailing that

there is. If your time is outside this then apologies but there will always be anomalies.

We are now putting PDF versions of C Vu on the website. These will be available in the second week of February, April, June, August, October, December. These are available to members only. Go to the main page <http://www.accu.org> and see the link on the first line under the 'General' section.

Advertising

Pete Goodliffe <ads@accu.org>

As observant members will have noticed, Chris Lowe has come on board as an ACCU advertising officer. He's now pretty much taken over the whole task and is coping admirably.

Thanks are due to him for volunteering for this position. It's a largely unnoticed role, but of real importance to the future of our publications.

Standards Report

Lois Goldthwaite <standards@accu.org>

Members of the C and C++ committees will be going the extra mile for the sake of standards work in the coming year. The October meetings this fall will convene in Kona, Hawaii, and the sessions next spring are scheduled for Sydney, Australia. The committees, which meet in successive weeks, try to distribute meetings equally between North America and Europe. So, conceptually, Hawaii counts as North America in this framework and Sydney as a European venue!

Despite the exotic locations, committee members will not be enjoying a beach-and-bikini boondoggle, far from it. The schedule calls for eight hours a day of hard technical sessions, often with additional work booked into the evenings as well. And there is plenty of work to be tackled. The C++ committee is sifting proposals to extend the standard library with new functionality, and at the same time examining directions in which the language can evolve to meet the challenges of more ambitious paradigms, such as template metaprogramming, while at the same time making the language easier to learn and use.

Of all the countries who participate in these standards committees, the UK contingent are consistently the largest and most active, apart from members of the US standards organisation ANSI. This is despite the fact that UK members are nearly all self-funded, whereas the expenses of delegates from other countries are usually paid by their employers. ACCU has already done an excellent job of educating members about the importance of the C and C++ standards and the professional value of participating in standards work - what can we do to educate employers on how this work brings benefits to them which make supporting it a worthwhile investment?

If you would like to join one of the BSI panels, please write to

standards@accu.org
for more information.

Copyrights and Trade marks

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

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

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

Dialogue

Student Code Critique Competition

Prizes provided by Blackwells Bookshops & Addison-Wesley

Please note that participation in this competition is open to all members. The title reflects the fact that the code used is normally provided by a student as part of their course work.

This item is part of the Dialogue section of C Vu, which is intended to designate it as an item where reader interaction is particularly important. Readers' comments and criticisms of published entries are always welcome.

Student Code Critique 22

The Code

Please look at the following code that is just playing with container of objects.

```
#include<iostream>
#include<fstream>
#include<vector>
using namespace std;

class A {
public:
    A(int p1= 0) : x(p1) { cout<<"CTOR"<<endl; }
    A(const A& a) {
        x = a.x;
        cout<<"COPY CTOR"<<endl;
    }
    ~A() {
        cout << "DTOR: address = " << (long)this <<endl;
    }
    friend ostream&
        operator<<(ostream& out, const A& obj);
private:
    int x;
};
ostream& operator<<(ostream& out, const A& obj) {
    out << "x = "<<(obj.x)<<endl;
    return out;
}

int main() {
    vector<A> v;
    // Be inefficient, not production code
    for(int i=0; i<3; i++)
        v.push_back(A(i));
    // Lo behold, container of objects
    copy(v.begin(), v.end(),
        ostream_iterator<A>(cout, ""));
    cout << endl;
    A * pa = &(v.back());
    // pa points to address of reference
    // to the last element
    cout << (*pa); // info
    cout << "pa = "
        << (long)pa << endl; // address
    cout << "call pop_back()" <<endl;
    v.pop_back(); // Line A
    // destructor WILL destroy last element
    // Now pa points to deleted memory, should crash
    cout << (*pa); // Line B
    cout << "pa = " << (long)pa << endl; // address
    delete pa; // Line C
    return 0;
}
```

Lines A and C delete the same memory, which shouldn't be allowed. But program runs fine allowing dereferencing at line B. Why?

From Matthew Towler <towler@ccdc.cam.ac.uk>

There are a few misconceptions that lead to the behaviour seen, and a number of small style issues in this code. I will begin with the more important, larger issues.

The first comment in `main` "be inefficient, not production code" implies that test code should be deliberately inefficient, and on changing to production code everything should be rewritten with efficiency first. Code should be written to be simple and easily understood first, and then optimised if this is shown by measurement to be necessary.

The comment on lines 10 and 11 of `main` is misleading. Taking the address of a reference yields a pointer to the referenced item, rather than the "address of reference", which has no meaning in C++. A more accurate comment would be

```
// pa points to the last element
The next issue is on line A, but first an aside.
```

Objects are created in two stages, first raw memory is allocated, either on the stack or the heap (with `new`) – at this point it is just a collection of bytes. An object is then constructed into this memory by a constructor – this initialises the memory to create an object of the desired type.

Destroying objects is the reverse; first a destructor removes the object from the memory leaving again a block of raw bytes, which are then returned to the system. `delete`ing a pointer performs both of these steps, but note that the steps do not have to be combined.

Line A commands the container to remove the last object; this will call the object's destructor, but does not imply releasing the memory. This last stage depends on the implementation of the container, and is not normally of concern to the programmer. During this call either the container reallocates its storage, leaving `pa` pointing at now unused memory; or (and most likely given the observed behaviour) the memory is unaltered, so `pa` now points at uninitialised memory. Note that for vectors, adding or removing elements invalidates all iterators and pointers into the container. When using any container you should be aware of what operations invalidate iterators.

This means the second line of the comment after line A is incorrect, the memory may or may not have been deleted.

Line B is then accessing uninitialised memory, which will give what is referred to as 'undefined behaviour'. Undefined means exactly that, it could do anything ranging from nothing, to (as some are fond of stating) reformatting your hard disk, and anything in between. Sometimes the code will crash (usually with a core dump or halt in the debugger) but it is wrong to expect a noticeable failure.

The observed behaviour suggests the uninitialised memory that `pa` points to was not altered by the destruction, so the access to the memory works as if the object still existed – perfectly in line with 'undefined behaviour'

The next line works correctly as `pa` has not been altered.

Finally line C attempts to destroy the object referred to by `pa`. This will first call the destructor, which will do something undefined (in this case the trace output makes it appear to work); it then calls `operator delete`. Deleting a pointer not allocated with `new` (such as `pa`) results in further undefined behaviour. In this case the particular implementation of `delete` presumably realises this and does nothing observable.

As regards the programmer's perceived problem – deleting memory already deleted has undefined behaviour, there is nothing in the language that detects and disallows such attempts.

Fundamentally, the programmer needs to understand that a `vector<A>` is storing the `A` by value i.e. by storing copies of `A` objects. Not by allocating `As` on the heap. The container deals with the memory management aspects automatically.

Now the smaller coding style issues, IMHO all are generally considered bad practice and can either be taken as read or the explanations found elsewhere.

- Use of a default value in the constructor argument rather than a separate default constructor

- Lack of `explicit` keyword on a single argument constructor, especially one taking an `int`.
- Use of direct initialisation in `A`'s copy constructor, rather than an initialiser list.
- C style cast of `this` pointer to `long` when outputting. This is not portable, and is unnecessary as there will be a system supplied output operator for pointers.
- Use of `endl` everywhere rather than simply `'\n'`, this is not crucial for test code, but it is a bad habit to get in to.
- `A` has a copy constructor but no assignment operator, these functions should always appear in pairs.
- The second comment in `main` does not add anything to the understanding of the code. Comments should only be used to explain the higher level intent of the code or unusual or difficult to understand passages.
- Use of `i++` rather than the preferred `++i` in a loop, this is not so important for `ints` but it is a good habit to develop.
- It is worth making use of the separator argument in `ostream_iterator` (" " does nothing) otherwise the output runs together e.g. 3 elements with values 12, 3, 4 would be output as "1234".
- Calling code which might throw an exception (such as output operators) in a destructor is dangerous, it should be surrounded by a `try...catch` block to stop any exceptions escaping.

From Ruurd Pels <ruurd@tiscali.nl>

Oh, well. I might as well try this. Here it goes.

1. Layout.

Apart from the obvious editing to fit the source in the columns, I would like to implore anyone that source code is a document. And documents should be readable, preferably to others. The code under scrutiny is sloppy in layout and lacks to convey what the intention is. More tongue-in-cheek, I think we can leave the time that there were language implementations that only allowed one-character variable names safely behind us. In my opinion, verbosity is not a bad thing, on the other hand, sometimes I myself go overboard.

A few style pointers are in order:

- a) Use type names and variable names that make clear what they represent, so, here, `class Integer` instead of `class A`.
- b) If code is documented inline, I like to see the comment indented at the same level as the code that follows. Not doing so tends to blur the outline of the code and makes it harder to read.
- c) Last, but not least, some general remarks regarding the purpose of the program would be in order.

2. Idiom.

In 'Advanced C++ Styles And Idioms', Coplien tells us that if and when we do not intend to create types that are second grade citizens, we must adhere to the 'Orthodox Canonical Form'. This means that a default constructor, a copy constructor, an assignment operator and a destructor must be implemented. It is my tendency to explicitly specify all of them instead of relying on the compiler to generate the default ones for me, so I added the assignment operator. As for the destructor, I almost invariably make them virtual, just to prevent rude surprises if I derive another class from a class I created. Only in very special circumstances and duly commented in the source code I use a static destructor.

The `main` function usually is defined as:

```
int main(int argc, char* argv[])
```

but when not using those arguments, I would prefer to see:

```
int main(int, char*[])
```

but certainly not:

```
#pragma argsused
```

```
int main(int argc, char* argv[])
```

In the same vein, returning 0 at the end of `main` should be replaced with the more portable `EXIT_SUCCESS`.

[Please note that I would argue with almost all the above. Do not provide a virtual destructor unless you intend the class to be a base class – and think very carefully before deriving from a non-abstract base. I would prefer the signature `int main()` if command line arguments are not used because it is the long hallowed idiom to express that intent and `return 0` is just as portable as `return EXIT_SUCCESS`. FG]

With respect to the `ostream` friend, I have a few remarks. Even in the eye of friendship, I tend to use accessor methods to obtain the inner value from the object to be streamed, but there is another little idiom I would like to present in this matter. If `class A` would be the base class for other classes, it is beneficial to separate the actual streaming from the body of the friend function and add that functionality to a protected virtual `streamOn` method *[but why not make it virtual? FG]* This makes the task of streaming much easier, because it is only necessary to overload the `streamOn` function in derived classes instead of rewriting the `ostream` friend boilerplate over and over again, so:

```
void Integer::streamOn(ostream& ostr) {
    ostr << getValue();
}
ostream& operator<<(ostream& ostr,
                  Integer const & obj) {
    obj.streamOn(ostr);
    return ostr;
}
```

Last but not least, if streaming an object to an `ostream`, adding text or an end-of-line to the streaming operation is generally a bad plan, because the user of the class then cannot decide how he will do output formatting. Every time he uses the streaming friend, he gets the text and the newline 'for free', possibly at a time that is inconvenient to him. Furthermore, it is very easy to forget to consume the text and the newline when at a certain time a corresponding input stream operator is written. Realizing that an `ostream` does not necessarily mean that the content of an object is printed on a console is important in this matter.

In the `main` function, a template is instantiated. I know that this is trivial code in some respect, but I prefer to use `typedefs` to create 'new' types based on templates, and while I am at it, `typedef` the iterators as well. So instead of:

```
vector<A> v;
```

I would prefer to see:

```
typedef vector<Integer> IntegerVector;
typedef IntegerVector::iterator
    IntegerVectorIterator;
typedef IntegerVector::const_iterator
    IntegerVectorConstIterator;
```

[If you are going to use those names I can see very little benefit from using typedefs. In my opinion a typedef should be used to create a more descriptive name for a type. Yours do not do that so skip them. FG]

and subsequently:

```
IntegerVector integerVector;
```

In a more general case, this even collapses to for example:

```
typedef vector<Rule> Rules;
typedef Rules::iterator RulesIterator;
typedef Rules::const_iterator
    RulesConstIterator;
```

[Now I agree about the first, but again I think the other two add nothing. FG]

in which case it is quite easy to change the container type to something different, for example, if rules are all of a sudden uniquely identified, into:

```
typedef set<Rule> Rules;
```

without having to change much in any code using the `Rules` type.

[and that is a good reason for the first typedef but not for the typedef applied to the iterators. FG]

Call them style idiosyncracies, I have developed some I guess. I have the tendency to use 'that' for the parameter name if I need to refer to another value of the same type, at least in the copy constructors and the assignment operator. Also, in streaming friends, I tend to name the `ostream&` parameter as `ostr`.

[Yes we all have our personal styles but we should always examine naming conventions and ask if they add to the readability of the code. FG]

3. Intent.

From what I gather from the source code, this is an exercise in reconnoitring the innards of a vector and the memory allocation strategies of the underlying operating system. I don't have a problem with stretching computers and operating systems. However, this type of code could be an indication that the student does not properly understand the true nature of object oriented programming at large or STL container classes in particular. Halfway through the `main` function the line between the concept of container classes and the technique employed by it is crossed. At a certain point in time, he seems to want to treat the vector container as an array of `Integer` objects, misusing the fact that `vector::back()` returns a `const` reference to the object in the container in order to create a pointer to that object.

At a certain point, the student wants to inspect the address of the object and uses a cast to output the address pointed to by the pointer to `A`. The cast is superfluous, and also wrong. If the intent was to convert the address to an integer number, on 32-bit architectures at least the cast would have to be towards an `unsigned long`, but since the cast is superfluous, simply streaming out the pointer would have yielded the address pointed to in hexadecimal notation. The simple:

```
cout << pa << endl;
```

would have sufficed. This leads me to believe that the student should again review how pointers are treated in C++, especially in the face of passing them to an `ostream` object.

The statement:

```
A* pa = &(v.back());
```

leads me to believe that the student must be taught the concept of ownership. It should be made clear to him that an STL container has ownership of the objects stored in it and that access to the elements of the container should be done through the methods the container provides. Awareness of ownership problems will make him a better software engineer.

Last but not least, the question why the program does not crash on line `B` and why the double deletion seems to be allowed. The case probably is that a vector allocates an array of objects to accommodate its elements. This means that `vector<T>::pop_back()` indeed calls the destructor of the element, but does not remove the storage space associated with it. In combination with the fact that we are dealing with a very simple class stored in the container, this could mean that even if the destructor is called, the memory is still there and is still used by the container. If the stored object was more complex, for example referring to dynamically allocated memory, the dereferencing probably would have gone awry as well as the double deletion of the object. For what it is worth, I tried this under gcc 3.2, and the second time the destructor is called through `delete pa` it is not even executed.

4. Finally.

I sort of tried to recreate the same problem here. The main code is not very much better in terms of properly handling the container, however, I adjusted a number of stylish and idiomatic flaws:

```
// Student Code Critique 22, Critiqued
#include <iostream>
#include <fstream>
#include <vector>

using namespace std;

class Integer {
public:
    Integer(int theValue = 0);
    Integer(const Integer& that);
    Integer& operator=(const Integer& that);
    virtual ~Integer();
    int getValue() const;
    friend ostream& operator<<(ostream& ostr,
                               const Integer& obj);
protected:
    virtual void streamOn(ostream& ostr) const;
private:
    int value;
};
```

```
Integer::Integer(int theValue)
    : value(theValue){
    cout << "ctor(" << value << ")" << endl;
}

Integer::Integer(const Integer& that)
    : value(that.value){
    cout << "ctr(" << value << ")" << endl;
}

Integer& Integer::operator=(const
                               Integer& that){
    value = that.value;
    cout << "op=(" << value
          << ")" << endl;
    return *this;
}

Integer::~Integer() {
    cout << "dctor(" << this
          << ")" << endl;
}

int Integer::getValue() const {
    return value;
}

void Integer::streamOn(ostream& ostr) const {
    ostr << getValue();
}

ostream& operator<<(ostream& ostr,
                   const Integer& obj) {
    obj.printOn(ostr);
    return ostr;
}

typedef vector<Integer> IntegerVector;
typedef IntegerVector::iterator
    IntegerVectorIterator;
typedef IntegerVector::const_iterator
    IntegerVectorConstIterator;

int main() {
    IntegerVector integerVector;

    for(int i = 0; i < 3; i++)
        integerVector.push_back(Integer(i));

    Integer* integerpointer
        = &(integerVector.back());

    cout << "integer value = "
          << *integerpointer << endl;
    cout << "integer address = "
          << integerpointer << endl;
    cout << "call integerVector.pop_back()"
          << endl;

    integerVector.pop_back();

    cout << "integer value = "
          << *integerpointer << endl;
    cout << "integer address = "
          << integerpointer << endl;
    delete integerpointer;

    return EXIT_SUCCESS;
}
```

[I made some minor corrections to Ruurd's English and added rather more comments than I normally do somewhat in the style I would use were I conducting a seminar. I am aware that naming style is

both individual and varies from one national group to another. However names should add value and, in my opinion, overly long names work to destroy the shape of code. It is no accident that mathematicians prefer very short names. The skill is in finding the point of balance.

My final point is that while I agree with the provision of streaming functionality through a member function, I would make it `public` and abandon the `friend` declaration of the streaming operator `<<`. There is still a tendency to overuse `friend` declarations. FG]

The Winner of SCC 22

The editor's choice is:

Matthew Towler

Please email francis@robinton.demon.co.uk to arrange for your prize.

Student Code Critique 23

(Submissions to francis@robinton.demon.co.uk by Sept. 6th)

It is very easy for more experienced programmers to forget how completely confused a novice can become when asked to write a program for some simple task. I have picked the following out of my slush pile because it illustrates the problem at many different levels. It starts with what is a clear confusion (if you can have such an oxymoron) about the problem itself and then goes downhill from there.

What would you do to help this student with his programming? Note that this time the student knows enough about C++ to write a suitable program, so the problem is in the mind to the extent that I think he is even confused by what is needed as output.

I am working on a program that will tell someone how old they are in years, days and months. I.e. You are 27 years, 200 months, and 1500 days old.

```
#include <iostream>
using namespace std;
void main() {
    int currentyear = 2003;
    int dob = 0;
    int result = 0;
    int day = 0;
    int month = 0;
    cout << "Enter your year of birth: ";
    cin >> dob;
    result = dob - 2003;
    cout << "You are " << result
        << " years old" << endl;
    cout << "Enter you month of birth: ";
    cin >> month;
    for(result=0; result <=12; result++)
        if(result <= 12) result = month+12;
    cout << "You are " << result
        << " months old"<<endl;
    cout << "Enter you day of birth: ";
    cin >> day;
    for(result=0; result <= 365; result++)
        if(result<=365) result = day+365;
    cout <<"You are " << result
        << " days old"<<endl;
}
```

Of course remember to comment on the program defects but your main focus should be how to explain the need for clear thought when producing a solution.

The Wall

Letters to the Editor

Book of the Year

Dear ACCU

I thought I would write in and explain why in my opinion Francis' idea for a book of the year award elicited such a small response.

This year I have so far read:

Title	Publishing date
Mastering Regular Expressions	Jan 2002
More Exceptional C++	Jan 2002
Programming with Qt	March 2002

2002 was a bad year for reading technical books, I only read:

Exceptional C++	Dec 1999
Learning Python	March 1999

2001 was much better but I will not carry on as it is tedious.

I do not consider these lists very long, but in the eight years I have been a professional C/C++ programmer I have always found myself to be an avid reader in comparison with the majority of my peers.

I tend to read books as and when I have the time, or when they are relevant to my current work. Which means (as the above list testifies) the number I read in the year of publication is small – usually only one or at most two. In turn this leads me to believe that I cannot really form an opinion of the best book published in the last year.

If instead it was asked what was the best book on a particular subject (say C++, design, teaching programming, most useful day to day etc.) published in the last five years (the shelf life of the average computing book) I think it would elicit a much greater response. For instance it would allow the C++ category to include such modern classics as the "Effective" and "Exceptional" C++ series.

Best wishes

Matthew Towler

Thank you for writing. I too have trouble even remembering the year of publication of books I've read, and possibly the requirement

of having read enough new books in the last year does rule out many people (though some of us try to keep up with the latest literature as much as we can). It is a sad reflection on the state of the industry that many professional users of C, C++ and similar languages have read fewer than a handful of books in their careers; some good recommendations from mainstream practitioners (excluding for now those who do read many technical books each year) might be more valuable to the majority than recommendations of the latest, greatest books on cool things Andrei can do to bring compilers to tears. (But Andrei, if you're reading: please do continue to torture compilers, eventually the leading edge helps to advance the average level of programming. And it's fun between now and then.) James

Thanks for the Compliment

I've just received this month's copy of C Vu. You can imagine my surprise in seeing my name quoted in such glowing terms! Whilst it *is* nice to receive compliments, for which my thanks, I feel something of a fraud. After all, all I did was send a fellow student a spare book I had! I hope that Michael derives some benefit from it and that his circumstances improve soon. Believe me, I know what it's like to be out of work having been there myself at one time (and hopefully never again!).

I will end on one piece of advice for students who, like Michael and myself, are trying to learn C++. Take it slowly as it's far from easy: the Pascal, Modula-2 and old-fashioned C from my university days are a breeze by comparison!

Best Regards,

James Bannon

Thanks for the reply and sorry to single you out – don't let this stop you from random acts of kindness in the future, I'll leave you alone, I promise. In truth, I know that this wasn't a big deal: I would like to think that I might do the same, as would a number of other ACCU members I have the pleasure of knowing (and I'm sure many I do not). But the fact that it's not a big deal is sadly not as universal as we might like, so you'll just have to live with occasional compliments! James

Francis' Scribbles

by Francis Glassborow

Concerning Complexity

Recently there was a long thread about the issue of whether we should use more than one return statement in a function. I think the most important thing is that we should try to understand the issues.

It is easy to say 'functions should have at most one return statement'. Actually if they are truly functions they need at least one return statement because the correct computer science term for something that does not return something is 'procedure'. But let me put that to one side and focus on the concept of a function in the context of C and C++.

There is certainly a correlation between bugs and multiple returns. Functions with more than one return have statistically more bugs. However there is a similar correlation between bugs and number of statements in a function. Functions with many statements tend to have more bugs than those with only a few statements. Finally there is a correlation between number of statements and number of returns; functions with many statements also tend to have multiple returns.

What I am suggesting is that the fault may not lie with multiple returns but with writing complicated code. We should avoid writing code that conceptually has multiple points of return even if we hide that by introducing extra variables or structures. The problem lies not with the multiple returns but with code whose design has more than one logical exit point.

But sometimes an algorithm has logically more than a single exit point. In such cases jumping through hoops to hide this feature breaks what I consider to be the prime directive for programming: 'Code should be a clear expression of the solution of a problem.'

There is a second programming directive 'Stop when you know the answer.' Let me give a couple of examples. The first is admittedly contrived but only to the extent that I want some simple code to illustrate a general principle.

Stop When You Have the Answer

Suppose that you have a two dimensional array of `int` (perhaps one representing the colours of pixels in a window) and you want to discover if a specific value is used anywhere in the array. For convenience let me assume that the data is stored in a `std::vector<std::vector<int>>`.

Here is a simple function to supply that answer:

```
bool contains
(vector<vector> > const & array,
 int value) {
    int const rows(array.size());
    int const columns(array[0].size());
    for(int row(0); row != rows; ++row) {
        for(int col(0); col != columns; ++col) {
            if(array[row][col] == value) {
                return true;
            }
        }
    }
    return false;
}
```

Now if you are a believer that functions should never have more than a single return you have a problem because however you reorganise your code the requirement is for two distinct exit conditions. The only ways these can be combined in a single return statement require either continuing processing after you know the answer or increasing the perceived complexity of the code. Increasing perceived complexity is a dangerous game because it increases the probability of bugs.

Yes, I am entirely capable of writing a definition of `contains()` so that there is exactly one return-statement but the cost will be an extra variable (trivial) and a more complicated condition in each of the `for`-statements. I happen to think that that violates my prime directive; it makes it harder to see the code as a clear expression of the solution to the problem. To justify that breach I need some compensatory benefit. I remain unconvinced that there is any. But maybe you know otherwise.

A Clear Expression of the Solution

Here is a function from a program of mine that implements Dr Conway's Life automata game:

```
bool will_be_alive(life_universe const & data,
                  int i, int j) {
    int const diagonal_neighbours(
        data[i-1][j-1] + data[i-1][j+1] +
        data[i+1][j-1] + data[i+1][j+1]);
    int const orthogonal_neighbours(
        data[i-1][j]
        + data[i][j-1] + data[i][j+1] +
        data[i+1][j]);
    int const live_neighbours
        (diagonal_neighbours
         + orthogonal_neighbours);
    if(live_neighbours == 3) return true;
    if(live_neighbours == 2) return data[i][j];
    return false;
}
```

There are many things about this code that might be worth discussing. One of them is the elimination of magic expressions by using `const` qualified definitions to provide names for the computational steps. Another is that I have consciously avoided the temptation to write the evaluation of the live neighbours as a pair of nested loops. I want a clear expression of the solution to the problem. I think that my solution is sufficiently clear that even those who know nothing about Life will be able to work out the rule for determining the status of a cell in the next program cycle.

I have not avoided writing the evaluation as a pair of nested loops for efficiency purposes (though I might get that as well) but for clarity of expression. In addition, I have laid out the source for the first two definitions (`diagonal_neighbours` and `orthogonal_neighbours`) to assist the clarity and not because of the constraints of the column width of C Vu.

Now look at the final three statements in that definition. Of course I can write them with a single return-statement:

```
return(live_neighbours == 3) ? true
      : (live_neighbours == 2) ? data[i][j]
      : false;
```

I do not think that the use of the conditional operator adds anything to the clarity of the original. I would expect any reasonable compiler to generate identical code for a release version though perhaps not for a debug version.

Subjective v Objective

For the teacher or instructor a rule such as 'functions may only have one return-statement.' is easy to measure but it does not result in good code in and of itself. What the student needs is a clear understanding of quality and I am afraid that that is something much more subjective. Most of us recognise good quality code when we see it but I doubt that we can exactly pin down what it is. Perceived complexity is a function of many things (one of them being the individual reader). Consider the following functionally equivalent implementation of `will_be_alive()`:

```
bool wba(vector<bitset<512>> const & d,
         int x, int y) {
    int const nd(d[x-1][y-1] + d[x-1][y+1] +
                d[x+1][y-1] + d[x+1][y+1]);
    int const no(d[x-1][y] + d[x][y-1] +
                d[x][y+1] + d[x+1][y]);
    int const l(nd+no);
    if(l == 3) return true;
    if(l == 2) return d[x][y];
    return false;
}
```

in which I have limited changes to names and layout, isn't that code harder to follow? And what about:

```
bool will_be_alive(
    life_universe const & data,
    int i, int j) {
    int neighbours(-data[i][j]);
    for(x(i-1); x < i+2; ++x) {
        for(y(j-1); y < j+2; ++y) {
            neighbours += data[x][y];
        }
    }
    return(neighbours == 3) ? true
       : (neighbours == 2) ? data[i][j]
       : false;
}
```


Is that more or less complex? I know what I think, but if you think it is fine, consider how you would change the above code if the rule for life gave different weights to diagonal and orthogonal neighbours, or if the rule included orthogonal neighbours that were two cells away from the cell being considered. Those nested `for`-statements coupled with the odd initialiser for `neighbours` just add fog.

Being Helpful

One of the by-products of writing a book is to discover how helpful people can be. Of course I will be giving full credit to the main players in the book's acknowledgements section but as the printed copy won't be around till early December and its target readership has little if any overlap with the membership of ACCU I thought that I should take time out to express my gratitude to several people who have provided a lot of added value.

Because he is a fellow author, and most particularly because one of his books might be considered a competitor (Teach Yourself C++ 7ed) I have to give pride of place to Al Stevens, a well known columnist for Dr Dobbs Journal.

Not only has he made his beginners IDE (Quincy) available to all including rival authors but he also takes the trouble to maintain it and try to address any problems with it that are raised by its users. Giving a product away is one thing but adding in support is another. Both should be appreciated.

Then I emailed him to enquire about his recommendations for providing an install mechanism for a CD. Straight away he responded saying that I was welcome to use the source code provided on the CD that comes with his book. That source code provides a simple little installer and uninstaller that is exactly what computer tyros need. Al is clearly that exceptional author who tries to meet the needs of his readers. I may not agree with all that he writes and my book takes a very different tack from his but my life and the lives of my readers will have been made much easier by his generosity.

Then there is all the work that Garry Lancaster has contributed in refining the original Playpen concept and implementing it so effectively. Again, without Garry's work my book would be a much poorer product. When we have checked that there are no hidden surprises I will arrange for my Playpen Library to go on our website where it will be free for anyone to use. I think it will be useful to any novice who wants to experiment with graphics, a mouse and direct access to a keyboard. For the time being they will be restricted to MS Windows OSs.

However, as soon as I have the time to test it out I have an X Window/Posix version of Playpen, which should considerably extend the portability of Playpen based code. Jean-Marc Bourguet has done that work.

Then there is a little article written by James Holland about a flood-fill algorithm he originally used in Pascal. The algorithm dates from 1988, so that explains why I missed it as all my graphics algorithms come from earlier times. I hope the Editor gets to publish that article because I have a response lined up generalising James' contribution to work in full colour.

Help Needed

Before I can add Linux users to the target readership I still need a simple IDE for that platform. Note the word 'simple' because I consider that essential. Learning to program is intellectually demanding and so we should avoid adding anything that can possibly be avoided.

Perhaps you have some expertise at customising one of the emacs type editors so most of the functionality can be suppressed. Novices do not write highly complicated code where such things as resource editors, class browsers and the like prove helpful. They want to have automated project management. In other words they should be able to determine what source code files and libraries will be used and where the user written header files will be found. They need a single hotkey to compile a source code file, they need a single hotkey to build a project. Finally they need to be able to use the compiler generated error messages to locate the relevant point in their source code.

What I need is a simple coupling between a public domain editor and GCC (preferably 3.2 or later). I am sure there are many people out there for whom this would be a wet Friday afternoon job (i.e. effectively a no-brainer). The real problem is finding one who does not immediately want to add a whole bundle of goodies.

Learning C++ using an MS Windows OS?

I have no doubt that if that describes you that you should seriously consider using the combination of MinGW and Quincy. That combination will allow you to keep focused on you programming rather than juggling with a whole bundle of gadgets.

For those that do not know, MinGW stands for minimalist GCC for Windows and Quincy is Al Stevens' cat.

Problem 10

```
int foo(bool read_all) {
    if(read_all) {
        string line;
        getline(cin, line);
        return atoi(line); }
    else {
        int i;
        cin >> i;
        return i; }
}
```

There are quite a few things wrong with the above function definition. Assume that all the appropriate headers have been included, what particular feature of C++ input makes it completely unusable?

Commentary on Problem 9

```
int foo(int i) {return ++i;}
int bar(int i) {return i;}
int main() {
    int i(21);
    cout << (foo(i) + bar(i++));
}
```

That one gives us a straightforward case of undefined behaviour because the call to `foo()` reads the value of `i` for no other purpose than because it needs the value. On the other hand the call of `bar(i++)` both reads and writes `i`. The Standard very definitely states that such mixture of uses (a plain read and a distinct write) without an unavoidable sequence point results in undefined behaviour. Note that the rules change if `i` is a user defined type. That means that it can matter if a template such as `std::vector<>` uses a plain pointer as an iterator or not. For example:

```
std::vector<int> vec(10);
std::vector<int>::iterator i = vec.begin();
std::cout << *i + *i++;
```

may or may not have undefined behaviour. If `std::vector<int>::iterator` is a user defined type the above simply has unspecified behaviour (either of the two sub expressions can be evaluated first. But if plain pointers are used then the result is undefined behaviour.

```
int fooref(int &) {return ++i;}
int barref(int const &) {return i;}
int main() {
    int i(21);
    cout << (fooref(i) + barref(i++));
}
```

Notice the difference, the `fooref(i)` does not need to evaluate (read) `i`, it just binds the parameter to the argument. The internal increment is fine because it is protected by the sequence points after the evaluation of the arguments and before the return. `barref(i++)` binds an rvalue (so it reads `i`) to the `int const &` parameter. As the only unprotected read of `i` is for the purpose of determining what should be written by the post increment we no longer have undefined behaviour. I think that if you check it carefully you will also discover that the result is still unspecified because the result depends on which function is evaluated first.

Cryptic Clues for Prizes

This time there were several offers of solutions. Perhaps the older readers remember playing with their electronic calculators with seven segment displays. For the younger readers there was a fad in the seventies for making up calculations that resulted in displays that could be read like words (sometimes upside down). In case you did not work it out, the solution to *It looks like a call for help.* is 505.

There are many potential numbers that can be clued this way (upside down is more productive) and we sometimes have to tolerate mixtures of upper and lower case letters. So we have 'Almost a capital example of what might be sold on an Antipodean shore.' in six digits is 577345. You need to check how a seven-segment display handles 4 to understand that one.

I am sending (when he jogs my memory) a copy of the C++ Pocket Reference to Bob Adler whose response had the added value of referring to his days in the dojo.

Now if I gave you 'That foolish day' as a clue you might quite reasonably think of 1st April. But what might 'An English programmer gets pieces of eight for the day of fools.' give as a two digit number? As an added clue an American would get a three-digit answer.

Again I will give a small prize to someone who supplies the English answer.

Features

On Not Being a Software Engineer

Simon Sebright

I write this to share my experiences good and bad about being unemployed in the software engineering arena. It's not meant to be advice, I certainly can't claim that my strategies have been overly successful. But it might give some comfort to those in similar positions, or arm you with some information should the worst happen.

A word in the office

Thankfully, I didn't find out by text message. Well, as I got my mobile phone for job hunting, it wouldn't have been possible. My boss in the open plan office got an internal phone call (one ring, not two), very curt, as if pre-arranged. He then came over to my desk, a word in the oberfuehrer's office. Pound, pound went my heart. I know what this is. Keep calm, don't hit anyone. Take it well and make them feel sorry for you.

They gave me the minimum possible redundancy payment and asked me to work my notice. I was amazed how little I came out with. The government allows the one week's salary per year to be capped at £250. So, you guessed it, for my toils, my commitment over the last two and half years, I got five hundred quid. It's tax-free, hooray, that'll help pay the bills, then. I asked if I could have my PC, to which they replied that it was needed as a server in the fragile IT set up. I wasn't even out the door.

Up and down

As I went home early, I felt very free. Powerful. Perhaps I was trying to convince myself I had made the decision. My wife was away with the kids, and I decided not to tell her on the phone, so lived with my terrible secret for a couple of days. I didn't go in the next day, but had some correspondence to find out who else had gone. Unlike the previous round of redundancies, where newer recruits and dead wood had been pruned, this was real cost saving. I went because I was paid well. Why was that? Perhaps because I was more productive. Still, it was a numbers game and I had the wrong number. I felt some truth in people's voices when they expressed surprise that it was me, it gave me some comfort, I suppose.

An ex-colleague at an old company had drunk himself to death shortly after losing his job a year or two ago. So, I had a few compensation beers, but didn't get stuck in too much. Rather I felt the need to do something. So I did all the jobs which had been on the to-do list for ever.

I didn't actually work that much of my notice in the end, with the odd interview and kids having chicken pox. When I reached my official termination date after holiday pay, it was another year.

It's the time of the year, dear

Well, there's never a good time to have a baby, or lose your job. New Year is probably the worst, though. You have to wait for all the budgets and plans to get agreed, and people to start feeling busy after Christmas's excesses. I'd signed up to a couple of internet job sites and started getting a trickle of things which matched my ever-broadening criteria. It was pathetic.

Find your friends

The best early lead I had was from a friend, whose department was about to recruit, so I got in on the act early. I had one interview, and then another with a test. It was all looking rosy, and then the management decided to change their plan. So, I got the job that wasn't. Back to the drawing board

Agencies - your friend and yours

My company had been really, really helpful and given me a dozen agencies' email addresses. Half of which were out of date. Don't be deceived by initial pleasantries and banter. These guys are not working for you. They are working for themselves first, the client second, and you last. You only

hear from them when they reckon that you would be able to get them some money in terms of their slice of the pie. Agent. Think about the word. Estate Agent. That's about it.

So, I sent my details to them all. Of the ones which got through, I had a number of email or phone replies to confirm things. One or two said there "might" be opportunities in a couple of places. As I only needed one job, this was great. Then nothing happened for quite some time.

Some agents are more active, and trawl the internet sites, so I occasionally get a phone call out of the blue about some job in a far-flung corner of England, usually south east. They talk of relocation as if it's like going shopping or to the pictures. Given the shortage of jobs, I'm now looking at anything anywhere, so relocation is on the cards. Near the beginning of the process, an agent had contacted me enthusiastically about a job somewhere I didn't want to live. I declined interest, and being a salesman he gave me some banter to make me think I might not have another opportunity like this one. In some ways I regret not taking it up. On the other hand, had I taken the job (should I have been offered it), I'd probably be regretting it about now. Can't win.

Time of the year, again

April come she will. And the end of the financial year. Another doldrum. Still, thereafter I got two interviews in the space of a fortnight. The first was one of wishy-washy style where you have to stick your oar in to make any positive contribution to the meandering themes of discussion. One where you just don't know what they'll think of you. They give everyone an assignment to do. I spent three working days on it and sent in my solution - we had to write a plug-in for one of their products given a couple of examples and a huge document describing the COM interfaces. I produced something which pretty much met the requirements, although I don't think it did much to demonstrate how I might be a good team leader. From the quality of the examples, I think they were looking for a hacker. Didn't get that one. Neither did anybody else, apparently.

Mr Right

Which exemplifies what's happening everywhere. Employers know they've got us over a barrel. You first have to play buzzword assault course. This is where you jump through hoops to demonstrate that you have the right letters after your name (COM, STL, etc.). You also have to have masses of experience and be exactly the right kind of person. It's particularly bad when they are looking for replacements for staff moving on. It would be quicker to campaign to change the cloning laws for humans and grow a new employee than to wait for them to pick someone by themselves.

The second interview I mentioned earlier was for a contract. The agent had said there wouldn't be a test. Wrong! A written exercise, and half an hour or more of a barrage of technical questions from dead pan people, like being on Mastermind, only I didn't get to choose the specialist subject. Apparently, I did OK on the technical side, but they didn't take me because I hadn't any tibco experience. In fact I hadn't even experienced the word!

Filling time

I'm the sort of person who is not bored. I've got a house and family, so there's lots of time taken. I've also tried to do some web development, which has been a challenge given my limited experience and resources. I've battled with TCP/IP and the like to get a mini network here (my company did agree to give me an old manky test machine), but I couldn't get the Access OLEDB drivers to work properly. I've gone for PHP/MySQL in the end and am about to get stuck in. As I do such things, I realise that basic software engineering skills and project management skills are all that count. And I get frustrated when job adverts require that you are an Oracle or SQL Server guru. Why? Relational databases aren't difficult to understand. C++ on the other hand does need some expertise. Only no one seems to need C++, or they don't think they need that level of expertise. Some adverts want graduate programmers or people with two years' experience. Why? Because they are cheaper. But, I like to think that you get what you pay for.

[concluded at foot of next page]

ACCU Spring Conference 2003 Roundup

Thaddaeus Frogley

The ACCU Spring Conference 2003, incorporating the Python UK Conference, was held between the 2nd and 5th of April. This report covers only some of the 57+ sessions at the conference, which had 5 “tracks”, plus evening “birds of a feather” meetings, and covered such diverse topics as C, C++, Java, Python, Haskell, language neutral design, patterns and more.

This article should give you a taste of what I saw of the conference. Unfortunately, I can't be in more than one place at a time!

Wednesday 2nd April

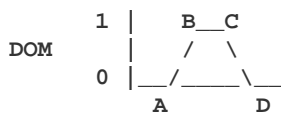
Linguistic Variables: Clear Thinking with Fuzzy Logic

Walter Banks, Byte Craft Limited

Walter Banks introduced the concept of a “linguistic variable” by way of example. A simple soup recipe was shown, and expressions such as “pinch” of salt highlighted. Walter pointed out that the real world is full of such imprecise terms, and that modelling them with a Degree Of Membership (DOM) value – which is normalised to a 0..1 range – can make working with them computationally and algorithmically simple and efficient. It is worth noting at this point that boolean values (`true` or `false`, 1 or 0) are a true subset of “fuzzy” DOM values.

Mapping a real world range of values, for instance temperature, to a linguistic variable is best achieved using a 4 point range graph, where the first value represents the point in the range below which the DOM is zero, the second value represents the point in the range up to which the DOM is between 0 and 1, and after which it is 1 (full membership). The third point on the graph is the point at which full membership ends, and the last point on the graph is the point after which the DOM once again becomes zero.

But a picture is worth a thousand words, so:



In which A, B, C, & D represent the first, second, third and fourth points described above, and the horizontal and vertical axis would represent a crisp real world value (such as temperature), and the Degree Of Membership respectively.

Audience members asked Walter if linear interpolation was enough, and he assured us that it was, and that despite there being corner anomalies with this approach, it has been shown in practice that the corner values are always non-critical.

Walter then went on to show how “fuzzy” logic operations can be implemented on DOM values very efficiently. A fuzzy `OR(a, b)` is

equivalent to a `MAX(a, b)`, and a fuzzy `AND(a, b)` is equivalent to a `MIN(a, b)`. Fuzzy `NOT(a)` is just `(1-a)`. Conditionals using linguistic variables were shown to work in a slightly surprising way. Any `IF` condition will boil down to a DOM value, which is then used as the degree to which the `THEN` expression is evaluated. This only really works well, as far as I can see, when the expression is a fuzzy assignment.

Commercial applications for these techniques include: electric motor starters, furnace control, aviation, loan application evaluation, fraud detection, stock price control, and of course, washing machines. Finally, it was interesting to learn that the 40,000 computer animated characters in the Lord Of The Rings movie, in the attack on Helms Deep scene, were controlled by a simulation system using approximately 135 fuzzy rules for each character type.

Coding Standards: Given the ANSI C Standard why do I still need a Coding Standard?

Randy Marques, Atos Origin

“There are 100 ways to do something, all equally good. Choose one, and stick to it”

Randy Marques gave a surprisingly entertaining talk on a difficult subject. He started the talk by pointing out that while the current ANSI C Standard is C99, when we talk about ANSI C almost everyone, including the majority of compilers think C89, causing many people to fall at the first hurdle – which ANSI C Standard? Will there be support for Variable Length Arrays? Incomplete Arrays? What about C++ style comments (`//`)?

The talk itself focused on the C89 standard. Appendix F of the C89 standard contains 267 items. Appendix F of the ANSI C Standard lists and describes all the unspecified, undefined, and implementation defined behaviour in the language (and a bit more). An example of unspecified behaviour would be the order of evaluation between sequence points. Many people know that: `array[i] = i++;` is asking for trouble, but how many also realise that the order in which the functions are called in this example: `i = f1() + f2() * f3();` is also unspecified?

Undefined behaviour (“dragons be here”) includes the exact behaviour in the case of integer overflow (wrap around, or saturate), and the behaviour of non-void functions with empty return statements. Implementation defined behaviour (consult your compiler documentation) includes the exact behaviour of casting a pointer type to an int, and the result of a right shift on a signed integer.

Randy also described several situations that can result in unexpected or unpredictable behaviour that are well defined in the standard, such as comparing floats for equality, and returning the address of a local variable.

Given all this, and many more reasons I do not have space here to go into, he argued, there is not only a strong motivation for development teams to adopt an internal coding standard, but for it to be enforced, where possible, with static (automated) testing – claiming that 40% of

When I was in work, I used to see things around me which might make a living. I've got a big garden, so thought I could dedicate a large area to propagating cuttings of my herbs. A rip off at £2 each on stalls, but I soon realised that I'd have to do hundreds and thousands to pay the mortgage. How much compost is that? I still think about becoming a plumber or carpenter. I can pretty much do it already given the work you need to do to get the typical British house remotely nice.

Doctor, doctor

My wife has been in academia for most of her life, and thinks I should become a professor, in German (she's German). I'd need to do a doctorate first. I sort of got persuaded and did some surfing of web sites I only half understood. I'm still not sure if I can do something which is interesting to me, meets research criteria and is actually useful in the real world, as I don't want to lose touch with things. I'd be keen to hear from anyone in academia, perhaps doing a PhD, or teaching something IT-related.

How long is a piece of job hunting?

Now and again the frustration of being out of work rears its head. Either I or the wife get stressed out. And usually one point which comes

forth is why I haven't done more. Simple, I haven't had time. You could spend all day, every day looking, but where? I feel I've done the 80/20 thing, and have covered most of the opportunities with my agency registration and email notifications. I don't know if this is correct, but given that the same jobs occur in the different systems, and sometimes the same job will appear twice in the same message through different agencies, I reckon I'm doing OK. To be honest, my best leads have come from being contacted by agents actively looking for skill sets, who think they have a “real” job vacancy to fill. As opposed to the one which took over three months to decide whether or not to interview me, and didn't, and then readvertised the job.

Conclusion

There's lots more I could write about each of the companies I've been to, or the way the agencies behave, or the helpful suggestions people make, but I hope to have captured here a flavour of what it's been like for me. Let's hope you don't get here, or if you are, that you get back soon! You've got to grin and bear it.

Simon Sebright

all runtime errors in C applications could have been found by using a static analysis tool.

When creating a coding standard, Randy told us, you will have to make some decisions on matters of style, in such cases his advice was: "There are 100 ways to do something, all equally good. Choose one, and stick to it. Do not try to make it a democratic process."

Other interesting facts in this talk were that any given bug fix has a 15% probability of introducing a new bug, and that the best fault rate in the world is that of the NASA engineers, who in production code, have a fault rate of "only" 6-8 faults per 1000 lines of code.

Randy Marques has kindly made the slides of this talk available from his homepage, at the following URL:

```
www.xs4all.nl/  
~rmarques/Werk/Pres/CodingStandards.ppt
```

C++ & Multimethods

Julian Smith

"C++ is deliberately designed to offer sharp tools when needed."

Julian introduced the audience to multimethods. Those of you who use the Dylan programming language or CLOS may already be familiar with, or just take for granted the existence of multimethods. Those of you that have had to implement the Visitor Pattern will be familiar with the problem that multimethods solve, if not the name.

Multimethods, it was explained, are methods that dispatch at runtime like virtual methods, but to more than one object. The virtual function call dispatch mechanism is a special case of the general multimethod mechanism, an example of "single dispatch", where the dispatch is determined by one object type. "Double dispatch" is the special case of method selection based on two objects. Multimethods generalise this to method selection on any number of object types.

An example application of multimethods is the double dispatch problem of deciding if two shapes overlap. An OO system might typically have a class hierarchy rooted with a Shape class, which might want to provide a public method for testing overlap, like so: "bool Overlap(Shape & a, Shape & b) { /* ... */ }", the problem comes when implementing this method, as it needs to know the derived type of both objects.

Multimethods, we were told, provide the solution. The multimethod mechanism presented was in the form of a language extension for C++, using a syntax previously suggested by Bjarne Stroustrup, which uses the virtual keyword as a type qualifier in the argument list of a non-member function like so: "bool Overlap(virtual Shape & a, virtual Shape & b);"

This function is declared by the programmer, and implemented at the compiler level, based on what type specific versions of the method have been provided. Julian also discussed techniques using multimethods to simplify GUI event handling and perform internationalisation of error messages.

Julian Smith has kindly made the slides of this talk available from his homepage, at the following URL:

```
www.op59.net/accu-2003-multimethods.html
```

An implementation of the language extension is available from the following URL

```
www.op59.net/cmm/readme.html
```

During the conference there was an interest expressed by compiler vendors in implementing this extension to the language. If this happens, and it gains more widespread usage, I would like to see it become part of the C++ Standard.

Thursday 3rd April

Keynote: In the Spirit Of C

Greg Colvin, Oracle Corporation

"Real programmers can write FORTRAN in any language."

Greg Colvin gave a genuinely engaging and provocative keynote to get the second day of the conference off to a good start. He focused on what he felt was the "spirit of C" linking C with C++ and Java, starting with a brief history of C, its origin, and the motivations that drove its designers.

Greg told us what he felt the spirit of C boiled down to the following key points:

1. Trust the programmer.
2. Don't prevent the programmer from doing what needs to be done.
3. Keep the language small and simple.
4. Provide only one way to do an operation.
5. Make it fast, even if it is not guaranteed to be portable.

He went on to explore how this "spirit" of C maps to C, C++, and Java as they currently stand. His angle on how C++ and Java map to the 5 "spirit of C" key points was that each language has adopted a focus on a subset of the 5 points, at the expense of the rest. For instance, C++ still holds the first rule in high regard, at the expense of the third rule, whereas Java places more emphasis on the third rule, at the expense of the 5th rule.

On C++ he said:

"There is no limit to the level of complexity that can be packed behind a beautifully elegant interface".

He observed that templates were added in order to support typesafe lists, and that the current trends in meta programming were entirely unexpected, accidental and impossible to prevent.

On Java he commented that the fact that there was no undefined behaviour in the Java language specification was "remarkable", but that didn't mean that you no longer had to trust the programmer, as threading was still hard, and deadlock common. He also noted that depending on automatic memory management can actually make it harder to manage memory.

Looking to the future, Greg urged that the C standards body "keep it real", and leave C99 as the final revision of the C standard. Of Java he said that true standardisation is needed, before the needs of the Java developer community are ignored by Sun in favour of corporate interests. C++, he said, should keep evolving, whilst being kept as close to being a proper superset of C as is possible.

Design & Implementation of the Boost Graph Library

Jeremy Siek, Indiana University Bloomington

Jeremy Siek started the talk off with a brief introduction to graph theory, explaining four commonly used graph-search algorithms: Breadth First Search (BFS), Depth First Search (DFS), Dijkstra's, and Prim's minimum spanning tree.

He went on to explore the commonality that these algorithms shared, observing that they all follow out edges, spread through the graph, and select from the visited unexpanded nodes to expand next. It was observed that the odd one out of the four, from an implementation point of view, was the DFS, and that by using generative configuration techniques the other three can all be implemented with a single function template interface; in the Boost Graph Library (BGL) that function is the `graph_search` function. Jeremy explained in detail the configurable elements of this interface, how they are used to implement the different search algorithms, and how the design decisions were arrived at.

The library makes heavy use of algorithm visitors in its design. For instance, the `graph_search` function template expects to be passed a Queue object, supporting `push`, `top` & `pop` methods, with the exact behaviour of the queue determining the behaviour of the search. Passing in a First In First Out (FIFO) queue results in BFS behaviour, a priority queue sorted on vertex distance results in an implementation of Dijkstra's algorithm, and a priority queue sorted on edge length implements Prim's.

One element of the talk that was of particular interest to me was the section on "breadcrumbs", which allows the user of the `graph_search` function template to provide their own system for recording node "colour" (visited and expanded, visited, or unvisited) by providing an object supporting the array syntax, indexed on node, which has the potential to be very efficient on trees that support marking at the node level, for example:

```
Colour& operator[] (Node& n) {return n.colour;}  
and at the same time allows searches to be performed on trees that don't support node marking by using an external "colour map".
```

The second half of the talk described the `adjacency_list` class template, which is highly configurable at compile time via the use of "generative" options, allowing the user to make their own decisions about functionality and implementation trade offs. An example of a functionality

trade off would be selecting a directed graph or an undirected graph. An example of an implementation trade off would be selecting to use arrays, or linked lists for the node “backbone”.

Asked by a member of the audience about the abstraction penalty, Jeremy claimed that it would only be apparent when using poor quality compilers, and that in theory using the BGL should be as efficient as custom code would be. I asked if BGL had an implementation of the A* algorithm, and was told that it did not. My impression was that the BGL is a very well designed, and carefully implemented library.

Secrets and Pitfalls of Templates

David Vandevoorde, Edison Design Group & Nico Josuttis

“Just get on with programming.” – David

“Don’t use templates.” – Nico

David (pronounced “Daveed”) and Nico are the collaborating authors on the “hot” new book: “C++ Templates: The Complete Guide”. That the book weighs in at 300 pages on a single language feature is a testament to exactly how difficult and complex C++ has become.

The talk went down well with the audience, and the joint presentation format worked well, with a witty interplay between David, a compiler implementer playing the expert, and Nico “playing” the role of the C++ template user. The first topic addressed dealt with the same terminology issues that were covered in Nico’s 2001 talk on template techniques. In addition it was explained that many people confuse “specialisation” (which is the result of instantiation) with “explicit specialisation” (which is where the programmer provides a different definition of a template for a specific type).

The talk then quickly moved on to some of the many “pitfalls” that can trip the unwary programmer, such as how scope affects name lookup. For instance, many people do not know that base class members are not automatically considered during name lookup when the base class depends on a class template parameter, leading to the recommendation that where this is desired the programmer use either:

```
Base : Foo();
```

or

```
this->Foo();
```

depending on the exact semantics desired. Another problem is that nondependent base members hide template parameters, that is, if you have a class template and it has a parameter that has the same name as typedef in a base class, then the template parameter is hidden by the name in the base class.

They looked quickly at template template parameters, noting that it was a core language feature added in order to support a library feature that no longer exists! Moving on to the Substitution Failure Is Not An Error (SFINAE) principle, which is essential for overloading function templates and currently a meta programming hot topic, they explained how SFINAE principal can be used to implement class templates for the automatic deduction of traits such as “is this type a class?”, or “does type Y have member X?”.

There were several other small tidbits that were interesting to learn during the course of the talk, such as the fact that Koenig look up is now properly called Argument Dependent Lookup (ADL) – perhaps because the standardisation committee wouldn’t want Andy Koenig to take all the blame! Another interesting thing I learned was just how expensive meta programming techniques can be at compile time, with one example given generating a whopping 3.5kb of symbol data per instantiation within the compiler (not however, causing bloat in the executable, as is commonly believed).

On `export` we learned that there is still only one implementation of it (the EDG front end), and that it may become an “optional” feature, or it may be dropped from the standard all together. It was noted that the “EDG” team’s implementation took 3 people 1 year part time to complete, compared to 1 person taking 2 weeks to implement template template parameters.

The quotes I gave under the heading for this talk refer to the replies given to an audience member who asked after the talk “So, given all these complications, what is your advice to programmers?”, Nico clarified his quick reply by saying that people shouldn’t “do stuff just because you can”, and David pointed out that a lot of the difficulties described during the talk really only affected people working in the corners of the language, and that most people will only ever need to know this stuff so that in the unusual case where things don’t work as expected, they know why they didn’t work as expected.

The Timing and Cost of Choices

Hubert Matthews, Oxyware Ltd

“Removal is a mugs game.”

Hubert Matthews opened his talk by asking the audience what they thought the talk would be about, based on its title. A mixture of answers were put forward, from compile-time vs run-time binding, to business level management decisions. Herbert told the audience that they were all right.

Most of the remainder of the talk focused on what Herbert described as the Choose-Check-Use pattern. This behavioural pattern describes the process and time line that occur when a decision is made. His claim was that decision making falls into the pattern of making a Choice, Checking it, and Using it, and that by recognising this pattern we are able to examine how the timing of these focal points allows us to further consider the cost implications. For instance, the longer the time between making the Choice and Checking it, the more time is wasted if the check fails. Likewise, the longer the Time Of Choice to Time Of Use (TOCTOU) the more risk that the conditions will have changed, and the check will have been redundant, forcing you to go back to the start and make another Choice.

Clearly this view supports the argument that you should not attempt to make a choice until the last minute, but with that view you have to be aware that work takes time, and that there may be dependencies on your decision. In such cases making an early decision can reduce “analysis paralysis” and let you “chip away” at the larger problem. Hurbert also noted that when it came to changing the time of a Choice it is harder to move it from late to early than from early to late.

He also described what he called the Prevention-Removal-Tolerance pattern in managing potential faults, both in programming and business decisions. The idea is that you prevent faults from getting into the system. The faults you cannot prevent, you remove, and the faults you cannot remove you tolerate. Hurbert recommended focusing on Prevention and Tolerance.

Hubert Matthews has kindly made the slides of this talk available from his homepage, at the following URL:

www.oxyware.com/Choices.pdf

Friday 4th April

Keynote: The Cost of C & C++ Compatibility

Andrew Koenig*, AT&T Shannon Laboratory

“C++ has become uncomfortably complicated”

Note: Andrew Koenig was unable to fly to the UK to attend the conference, and his slides were instead presented by conference organiser Francis Glassborow.

One of the first slides of the keynote read “The opinions expressed in this presentation are not necessarily those of the author”, which, under the circumstances, got a good laugh.

The focus of the talk was to be the balance of stability vs stagnation. The slides told how he feels that if C++ sticks too close to the past (i.e. source code compatibility with C) it risks becoming marginalized in the future.

The presentation described the changing face of computing: How CPU performance increased, even outpacing memory and I/O, resulting in low level performance becoming less important than it used to be. Programs can now do more in less time, usually the bottleneck is bandwidth, be it the bus, hard drive, or network. This is allowing interpreted and bytecode interpreted languages to flourish. With, for example, Java on the client-side, Perl on the server, and C# for windows applications, what place for C++?

The talk also explored the implications and consequences of the C compilation, linking, and execution models, and runtime library that C++ are tied to, comparing them to what is now being done by languages choosing not to tie themselves to a legacy language.

Andrew Koenig feels that it is time that the C & C++ communities acknowledge that C and C++ are two different syntactic bindings to a common semantic core, and define the nature of that core, allowing new bindings to it that are recognisably members of the C & C++ family, incorporate all the good stuff from C++, and from other languages as well, and omit as many of C++’s present problems as possible.

Double session:

Advanced Template (and namespace) Techniques

Herb Sutter

Herb Sutter gave an in-depth talk on the real-world, practical issues in using C++ templates, and namespaces. He rooted the theory firmly in the real world by way of live compiler comparisons, showing what a selection of compiler and library combinations do and don't do in practice. The talk focused primarily on the following questions: What are dependent names? What is two phase name look-up? How does unwanted ADL affect your existing templates, and how can you avoid these problems? How should you enable users to customize your template(s)?

When you write a template, we were told, any name that depends on a template parameter is a dependent name. That is, any qualified or unqualified name that explicitly mentions a template parameter, any name qualified by a member access operator with a dependent type on the left hand side, and any function or functor call which has any arguments that are of a dependent type, that is not qualified with a non-dependent type.

Two phase name lookup splits the time during compilation where names are looked up into two phases. The first phase is at the point of definition, the point where the template's actual definition is parsed. This is when all the non-dependent names are looked up. The second phase is at the point of instantiation, this is the point at which the template is instantiated, creating a specialisation for actual types. This is when the remaining (dependent) names are looked up.

It should be noted at this point that most compilers currently do not implement two phase name lookup, and those that do may hide it behind obscure non-default command-line switches. When, in 1997 HP implemented it they discovered that could not find a single piece of non-trivial template code that was not affected by it, either at compile time or runtime. This raises the question: Why have two phase name lookup if it will have such a huge impact on existing code? The main reason for it is "template hygiene". Two phase name lookup allows the template writer to better isolate their code from contamination by client code.

So what can the template author do to stop client code hijacking his names? One workable solution is to make use of namespaces for your code, and explicitly qualify any potentially dependent names that you don't want to be used as a point of customisation as coming from that namespace. This will work now with nonconforming compilers, and will also continue to work in the future when compilers implementing two phase name lookup become common (i.e. GCC 3.4 will have it).

ADL kicks in when the compiler encounters an unqualified name that is followed by a list of arguments in parentheses, or an overloaded operator is called using operator notation. When this occurs the compiler also looks in the associated namespaces and classes of each of argument's types. ADL is suppressed when the name is qualified, or ordinary lookup finds the name of a member function (with the exception of operator overloading member functions). All this raises the question: Why have ADL if it is so complicated? The main reason for it is to simplify the use of operators when used in conjunction with namespaces. An example of this is the use of C++ IO streaming operators in the `std` namespace. In addition, ADL implements the interface principle described by Herb Sutter in his ACCU 2001 talk.

A significant chunk of the second half of the double session explored an apparently simple, well formed code sample that caused problems on many compiler & library implementations due to combinations of unplanned for ADL, and missing two-phase name lookup hijacking names hidden in apparently reasonable library implementation code. Two ways of "turning off" ADL were then shown. The first is to explicitly qualify the namespace of the function call, which is a good solution for most, but not all, situations. The second is to use the function pointer syntax trick, sometimes used to avoid macros in the C library.

The talk then went on to go into detailed advice for library implementers, with regards the use of namespaces and templates, but I think that this article is becoming too long already, so I'll not go into the details of it here, but it boiled down to basically the same advice given above for dealing with two phase name lookup.

Herb then went on to give his advice on providing points of customisation in template code. What the template implementer should

do at points of customisation is make unqualified calls to the customisable name, whilst explicitly suppressing ADL on all other names, and the library user can customise by writing their own version (that uses the type used to instantiate the template) in their own namespace (the one containing the type used to instantiate the template). Another option is to use explicit specialisations. The template implementer provides a default class template in the library namespace and makes qualified calls to it. The library user writes their own explicit specialisation of this class template and places it into the library namespace. Whichever option is chosen, the library implementer should always document it.

Saturday 5th April

Keynote:

The Internet, Software and Computers – A Report Card

Alan Lenton, Interactive Broadcasting Ltd

A bleary eyed Alan Lenton opened the last day of the conference with an interesting, if slightly politicised talk on the state of the IT industry as a whole. He started the talk by pointing out two trends in programming language development:

1. The rise of high-level interpreted languages, e.g. Python, Ruby
2. The rise of "commercial" languages, e.g. Java, C#

He argued that that, whereas non-commercial languages were designed to fill a hole, or scratch an itch, the "commercial" languages were designed to fulfil some commercial interest. In the case of Java, that interest was an attempt by Sun to marginalize Microsoft. In the case of C#, that interest was for Microsoft (to fix holes in the OS API, but also) to counter Sun.

Alan then asked if C++, being both a high level and low level language, and non-commercial, was in danger of becoming thought of as the "sofa-bed" of programming languages, observing that sofa-beds don't make good sofas, and that they don't make good beds either!

Alan then went on to consider the issue of "who owns your computer?" – with software patents (SCO suing IBM over Linux), End User Licence Agreements (EULA), software rental models, and Palladium, should we take it that big business regrets the general purpose personal computer? There is a battle for control going on that extends beyond even this, with people asking: "In the future will the US own the Internet?" and "Who will own the DNS servers?" There is a demonization of the internet (child porn, etc), and yet at the same time a real need to deal with genuine social problems arising from widespread internet use, such as spam and viruses. He also observed that people want to use the internet as a panacea for education, and the rising trend of people assuming they have a right to have access to "free" stuff online – observing that it is always paid for by someone, and that during the dot com boom that just happened to be venture capitalist money.

Another trend highlighted by Alan's keynote was the politicisation of technology, citing the Regulation of Investigatory Powers (RIP) Act 2000 in the UK, and "eGovernment". With internet usage currently static at around 60%, Alan asked, is "eGovernment" doomed to create an underclass of "unwired"?

Unifying these disparate points was the idea that: "The innovation that that accompanied the rise of first the personal computer, then later the internet, and which hasn't yet finished, destroyed the marketing model of a number of powerful interests. Since they are entrenched and powerful, they are trying, at a number of levels, to fight back through the legislative and judicial systems."

Multi-Platform Software Development: Lessons from the Boost libraries

Beman Dawes, StreetQuick Software

"Test early, test often."

Beman Dawes presented a talk aimed at an introductory and intermediate level audience (both coders and managers), drawing on his experience in the development of Boost, a multi-platform library that aims to work on all systems that support C++ – from the largest Cray mainframe to the smallest embedded system.

The talk first explored the issue of what counts as a platform, listing not just operating systems as platforms, but also OS versions,

compilers, tool-sets, internationalisation, hardware configurations – including “compatible” hardware with different performance characteristics, and office environments (culture clash) as containing potential “platform” issues. Beman suggests that the traditional “port”, where you develop on X now, and port to Y later is asking for disaster – showing a photograph of a warning sign that reads “Beware of the Wildlife” as a visual device.

The main thrust of Beman’s multi-platform strategy was test-focused, advocating a policy of “Test early, test often.”, stressing automated testing, at least once a day. He also advocated such test-focused policies such as adding test cases before fixing bugs – and making sure the un-fixed code fails the test case before attempting to fix the problem. Not that this isn’t quite the same as the unit-test-first policy advocated by the XP community, which advocates creating the test cases before implementing the code. Beman also suggested a policy of test result publication within the whole project team, as an aid to “team psychology” (a term he used no less than three times during the talk).

Beman acknowledged the harsh reality of software development by suggesting that project teams use “surrogates” for platforms that they “might” have to target, but do not (“currently”) have access to, for instance using compilers that are available on the missing platform, or using a free UNIX such as Linux or BSD as a surrogate for a commercial unix such as Solaris or SCO.

He introduced the audience to what he called the Unified-Platform Development Process (UPDP), which places focus on having a single codebase and repository regardless of platform, with “single codebase” including source code, build scripts, test scripts and data, documentation, etc. The audience were also encouraged to use Platform Neutral Development Tools (PNDT), which should “work on one, work on all” – stressing that these tools should not only be portable in and of themselves, but should also have portable inputs, be sufficiently powerful to do the job, and be culturally acceptable to the project team. PNNTs might include: repository/SCM clients, compilers, libraries, build systems, test systems, scripting languages, documentation tools, and defect tracking / task allocation software.

Beman said that by investing the time up front to find PNNTs and set up a UPDP the benefits of such a system are: multiplatform failures detected early, developers don’t need to be experts on every platform, the cost of adding additional platforms is lowered, and improved project psychology.

The tools described in the talk are available from the following URLs:

www.boost.org/tools

www.boost.org/libs/config

www.boost.org/libs/compatibility

Note however that these represent one of many options. Other free and commercial tools exist, and each team should decide which platform neutral tools are best for their project.

Pattern Writing: Live and Direct (double session)

Kevlin Henney, Curbralan Limited & Frank Buschmann

“The class is not a useful element of design ... :)” – Kevlin

Kevlin Henney and Frank Buschmann’s joint presentation on pattern writing was interesting and thought provoking, though it did lack the natural ‘chemistry’ in the back-and-forth between the two speakers as was evident in David Vandevoorde and Nico Josuttis’ Secrets and Pitfalls of Templates talk.

The talk started by introducing Patterns, citing several different views on the subject, the best of which being, in my opinion, a quote from Jim Coplien which states that

“a pattern is a piece of literature that describes a design problem and a general solution for the problem in a particular context”.

The talk explored how patterns document recurring design, and apply to domains outside of programming, for instance, architectural, or organisational patterns. Thus design patterns are sensitive to context. Patterns are not something that you “add” to a project, in the same way as “quality” isn’t something that can be “added” to a project. Patterns document an design story, from problem to solution, with consequences and rationale. It must document what, how and why.

Many people seem to think that the “Gang of Four” (GoF) book is the only, or definitive, book on software patterns. It is a good book, we were

told, but it is 10 years old now, and the industry and its understanding of design patterns has moved on.

Kevlin and Frank went on to talk about the essential elements of pattern documentation, making repeated comparisons with GoF. They listed and explained the following sections as vital to pattern writing: Identification (name), Context (where the problem occurs), Problem, Solution (not necessarily a class diagram), & Consequences (design is always a trade off). They also recommended providing motivating examples but not a reference implementation.

They also talked about grouping patterns into pattern communities and pattern catalogues, and how some patterns will be complementary, meaning in this context, opposites, completing a natural symmetry of alternative solutions to a single problem, with balanced trade offs. Pattern decomposition was also examined, with the GoF Singleton decomposed into 5 separate patterns by way of example.

The first half of the double session closed with an examination of pattern languages, which forms a vocabulary that connects pattern communities, allowing pattern solutions to be described in terms of other patterns, with context equally if not more important than it is with individual patterns.

The second half of the talk was in the form of an interactive/audience participation pattern-writing exercise in which an Object Manager pattern, dealing with object creation (Factory Pattern) and life time management, and destruction (Disposal Methods) were discussed and documented, putting the theory of the first half of the talk into a more practical context.

Kevlin has kindly made the slides of this talk available from his homepage, at the following URL:

www.two-sdg.demon.co.uk/curbralan/

papers/accu/PatternWriting.pdf

End note, extras, etc.

Standardisation, Organisers & Sponsors

The ACCU Spring Conference is organised by Desktop Associates Ltd, in conjunction with the ACCU. One of the activities the ACCU is involved in is the support of standardisation work. The ACCU 2003 Spring Conference was organised as to be at the same venue, at the same time as the WG14, WG21 & J16 standardisation meetings, a practice that ensured the highest quality of speakers and delegates (Bjarne Stroustrup, for example, attended). I think it is worth recognising the sponsors who provide financial support for the standardisation meetings. This year those sponsors were: Microsoft, LDRA, Intel and Hitex. In addition, the conference itself was also sponsored by Perforce Software, and Blackwells. For the sake of completeness, also in attendance as exhibitors were: Rogue Wave Software, QBS Software, and PCG.

Slides & Papers online

Some of the speakers have made the slides and/or papers from their talks available online. Here is a selection of the ones from talks I was unable to attend:

Kevlin Henney’s “C++ Threading.”

www.two-sdg.demon.co.uk/curbralan/

papers/accu/C++Threading.pdf

Jon Jagger’s “Sauce: An OO recursive descent parser; its design and implementation.”

www.jaggersoft.com/sauce/

Mark Radford’s “Pattern Experiences in C++”

www.twonine.demon.co.uk/articles/

Pattern-Experiences-in-C++.zip

Thaddaeus Frogley

References and useful links

[1] If you enjoyed this article, you may also enjoy the write up of the ACCU Spring Conference 2001, by the same author:

http://thad.notagoth.org/accu_spring_2001/

[2] ACCU website: <http://www.accu.org/>

[3] Python UK website: <http://www.python-uk.org/>

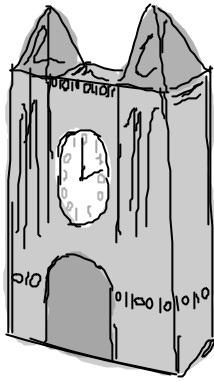
Professionalism in Programming #21

Software architecture

Pete Goodliffe <pete@cthree.org>

Ignorance transcends architecture

James Gaskin



Go into a city. Stand in the middle of it. Look around. Unless you've picked a pretty unusual place you are surrounded by a large number of buildings of varying ages and styles of construction. Some fit in to their surroundings sympathetically. Others look totally out of place. Some are aesthetically pleasing and seem well proportioned. Some are downright ugly. Some will still be there in 100 years time. Many will not.

The architects that designed these buildings took a lot into consideration before they even put pen to paper. Then during the process of design they worked carefully and methodically to ensure the building was feasible to fabricate, and balanced all the contending forces; user requirements, construction methods, maintainability, aesthetics, and so on.

Software is not made of bricks and mortar, but the same sort of careful thought is required to ensure a system meets similar sets of requirements. We have been erecting buildings far longer than we've been writing software, and it shows. We're still learning about what makes good software architecture.

Not too long ago I started working on a project whose software was pretty much undocumented and had been allowed to develop wildly with no careful green-fingered gardener to prune and tend the borders. Naturally it had become a woolly mess. Time came when we needed to understand how it all *really* worked, and an architectural diagram of the system was drawn up. There were so many different components (many largely redundant), inappropriate interconnections, and different methods of communication, that the diagram was an intense jumble of tightly woven lines in many interpretive colours. Almost as if a spider had fallen into a few pots of poster paint, and then spun psychedelic webs across the office.

And then it struck me. We had all but drawn a map of the London Underground. Our system bore such a striking resemblance it was uncanny – it was practically incomprehensible to an outsider, with many routes to achieve the same end, and the plan was still a gross simplification of reality. This was the kind of system that would vex a travelling salesman.

The lack of architectural vision had clearly made its mark on the software. It was hard to work with and hard to understand, with bits of functionality strewn across completely random modules. It had got to the point where the only useful thing you could do with it was throw it away.

In software construction, as in building construction, the *architecture really matters*.

In this little foray into the world of software architecture, we'll take a look at what it really is, what it really isn't, what it's used for, and investigate some common architectural patterns.

What is software architecture and what good is it, anyway?

Architecture is the art of how to waste space

Philip Johnson

So is this just another term that stretches the “building” metaphor a little thinner? Maybe so, but it is a genuinely useful concept. Software architecture is sometimes known as *high-level design*; terms get mixed up, but the meaning is the same. Architecture is just a more evocative description of this concept.

What is it?

As an architect prepares a blueprint for a building, so the architecture is a blueprint for the software system. However, whilst a building's blueprint is a rigorously detailed plan with all the important features included, our software architecture is a top-level definition, an overview of the system specifically avoiding too much detail. It's macro, not micro.

In this high-level view all implementation details are hidden, and we just see the essential internal structure of the software. The architecture identifies the key software modules (or components, or libraries, at this point call them what you like: *blobs*), and identifies which ones

communicate with each other. The architecture helps to identify and determine the nature of all the important interfaces in the system and helps to clarify the correct roles and responsibilities of the various subsystems. This information allows us to reason about the system as a whole without understanding how every part will work.

The key is that this design should be *simple*. A few well-chosen modules and sensible communication paths are the aim. It also needs to be comprehensible, which often means visually represented. A picture speaks a thousand words, after all.

In this way, the architecture is a framework into which the real development fits. It allows further design work to proceed with our focus on the right parts, and provides an initial way to split up work between teams. It allows us to weigh up different

implementation strategies, in the same way you'd plan a journey from your home to a holiday destination in a different continent.

Not only does the architecture give a picture of how the system is composed, it also shows how it should be extended over time. In a large team, programs develop more elegantly when there's a clear unified vision of how the software should be adapted, what should be put in each module, where later modules will fit in to place, etc.

Exactly what needs to be addressed by the overall architecture will differ from project to project. On the whole, the target platform is not all that important at this stage; it may be possible to implement the architecture on a number of different machines using different languages and technologies. However, for certain projects it may be important to specify particular hardware components, most likely for embedded designs. For a distributed system the number of machines/processors and the split of work between them may be an architectural issue. If it's really fundamental to the overall design the architecture may also describe specific algorithms or data structures, although these two are less likely. There is a trade-off, though. The more information that gets set in stone at the architectural level, the less room for manoeuvre there is at a later implementation stage.

In physical architecture we use a number of different drawings or views of the same building: one for the physical structure, one for the wiring, one for the plumbing, etc. Similarly we may develop different software views in the architectural process. Four views are commonly recognised:

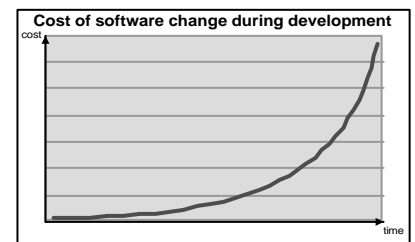
- **conceptual view** (or logical view), shows the major parts of the system and their interconnections,
- **implementation view**, a view in terms of the real implementation modules, which may have to differ from the neat conceptual model,
- **process view**, the dynamic structure in terms of tasks, processes, and communication, used if there's a high degree of concurrency involved,
- **deployment view**, shows the allocation of tasks to physical nodes, in a distributed system.

These views may arise as development work progresses. The main result of the architectural process is the first view, and that's what we're concentrating on here.

Where and when do you do it?

The architecture is captured in a high level document called something imaginative like the *Architecture Specification*. This specification explains the structure and shows how it fulfills the system requirements, including important issues like performance requirements, and how acceptable fault tolerance will be achieved.

It is therefore the first developmental step after the requirements have been agreed upon. It's important that this is done up front because it provides a first chance to validate design decisions that will have the most significant impact on the project. It will expose weaknesses and potential problems, saving a lot of time, effort and money if a bad decision is reversed this early on. It's expensive to change the foundation under a system when a lot of it has been built. As the graph illustrates, the cost of a fixing a problem escalates exponentially as you defer it!



1 OK, so the graph I present here is somewhat hand-wavy, I don't have space to discuss figures. It has been adapted from hard data presented in several different places, for example Boehm's investigations in [1].

Certainly, architectural work is a form of design, but it is separate from the design phase, and distinct from low level design work, although it certainly overlaps somewhat. Later work on detailed design may feed changes back up to the system architecture. This is natural and healthy.

What is good architecture?

In a well-designed system there should be neither too few nor too many components. Of course what this means differs from project to project, get a sense of scale here. For a small program the architecture may fit on (or be done on) the back of an envelope, with just a few modules and some simple interconnections. A large system naturally requires more effort, and more paper.

With too many fine-grained components, the architecture is bewildering and hard to work with. It would imply that the architecture has gone into too much detail, and has become more of a general design. If there are too few components then you see far too much work being done by a single module. This makes the structure unclear, hard to maintain and hard to add to. The correct balance is in there somewhere.

The architecture should give as little information as possible about the inner workings of each module. The goal is that each module shouldn't have to know much about the other parts of the system. We aim for high cohesion and low coupling at this level of design, as with all others.

The architecture specification shows the set of design decisions made, and makes it clear why this approach is being favoured instead of the alternative strategies. It doesn't need to labour these other approaches, but must justify the chosen architecture and prove that some thought went into alternatives. It should have identified the primary goal of the architecture – for example modifiability is different from performance, and will lead to different architectural design decisions.

A good architecture leaves room for manoeuvre, it allows you to change your mind. For example it may specify that we wrap third party components with abstract interfaces so we can swap one version out for another. It may suggest technologies that make it easy to select different implementations during deployment.

The architecture must be clear and unambiguous. We should favour existing well-known architectural styles (see the later section for more on these), or should use known frameworks. It should be easy to understand and start working with.

A good architecture has a certain aesthetic appeal that makes it *feel* right.

What is the architecture used for?

Obviously, it's a key part of the system design. But the architecture has a number of uses that stretch a little further than this. We use the system architecture to:

1 **Validate.** As we've seen, the architecture is our first chance to validate what is going to be built. We can check that the system will meet all requirements. We can check that it really is feasible to build the system. We can ensure the design is internally consistent and hangs together well, with no special cases or gratuitous hacks. Nasty blemishes in the high-level design will only lead to more nefarious hacks at lower levels. The architecture helps to ensure we prevent any duplication of work, wasted effort, and redundancy. We use it to check there are no gaps in the strategy, that we have included all the necessary pieces. We ensure that there will be no mismatches as separate sections are brought together.

2 **Communicate.** We use the architecture specification to communicate the design to all interested parties. These may be system designers, implementers, maintainers, testers, customers, or managers. It's the primary route to understand the system, and as such an important piece of documentation should *always* be kept up to date as changes are made. Like any other piece of documentation, it can become dangerous as a lie.

The architecture conveys the vision of the system. It should identify how future extensions fit in neatly. It helps to maintain the "conceptual integrity" of the system, which Brooks speaks of [2]. It implicitly provides a set of conventions, and contains an element of style. For example, it would be clear that you shouldn't introduce a custom socket connection for new component's communication if the rest of the design uses a CORBA infrastructure.

The architecture should naturally provide a route into the next level of design without being prescriptive.

3 **Discriminate.** We use the architecture to help us make decisions. For example, it identifies build vs buy decisions, identifies whether usage of a database is necessary, and clarifies the error handling strategy. It will flag problem areas, the areas of particular risk on the project, and help us plan to minimise this risk. Just as an architect's primary goal is to check his building stays up when it's built, under all expected conditions (and some unusual conditions too), so should our software structure produce a resilient product. A little wind or extra load shouldn't topple the thing over.

We need this system-wide perspective to make the appropriate tradeoffs ensuring the design meets its required properties. These important points are considered at the beginning rather than grafted in towards the end of development.

What do we determine about the components?

A good architecture captures information about each component, whatever *component* means in the architecture's context. It could be an object, a process, a library, a database, or a third party product. Each of the system's components will be a clear and logical unit. They each perform one task, and do it well. No component includes a kitchen sink, unless there's a specific kitchen sink module.

Whilst it won't dwell on a module's implementation issues, the architecture will describe any exposed facilities, and perhaps the important externally visible interfaces. It defines the visibility of the component, that is what it can see, what can see it, and what can't. Different architectural styles imply different visibility rules, as we'll see in the next section.

What do we determine about the connections?

A connection may be a simple function call, or data flow through a pipe, it may be an event handler, or a message passing through some OS/network mechanism. A connection may be synchronous or asynchronous. Some collaborating components may have certain shared resources that they communicate indirectly through (for example, a subordinate component, a shared memory region, or something as basic as a file).

The architecture identifies all the inter-component connections, and describes all relevant connection properties. A property is relevant if it impacts how the system will operate.

Architectural styles

Form ever follows function

Louis Henry Sullivan

Just as an immense gothic cathedral and a quaint Victorian chapel, an imposing tower block and a 1970s public lavatory employ different architectural styles, there are a number of recognised software architectural styles that a system may be built upon. These styles are chosen for various reasons, good and bad, and differ in several ways. For example:

- How resilient they are to changes in the data representation. In the worst case every component may need changing (e.g., a pipe and filters architecture). You might discover the need to change data representation when the input becomes too large to be held in memory at once.
- How resilient they are to changes in algorithms.
- How resilient they are to changes in functionality.
- The method of separation/connection of the modules.
- Their comprehensibility.
- Their accommodation of performance requirements.
- Any consideration of reusability of components.

In practice we may see a mixture of architectural styles in one system. Some data processing may progress through a pipe and filter process whilst the rest of the system's control is through component based design.

The following sections describe some common architectural styles. And compare them to pasta. Don't ask why.

No architecture

Like my London Underground project, a system never has *no* architecture, just no *planned* architecture. Before long this state of affairs becomes an albatross around the neck of your development team. The resultant software *will* be a mess.

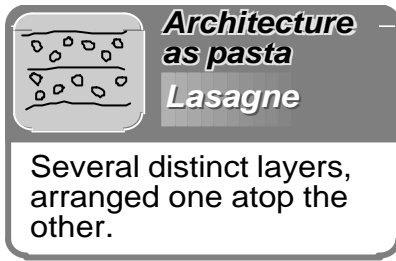
Architecture as pasta
Spaghetti Ball

Messy, uncontrollable, unmanagable, morass of interwoven loop.

Architecture is a requirement of building good software. It is a part of the development process. Not planning an architecture is a sure fire way to doom yourself before you've even started.

Layered architecture

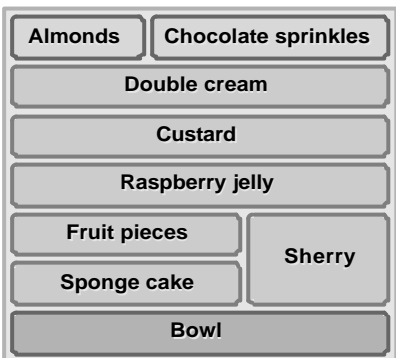
This is probably the most commonly used architectural style in conceptual views. It describes the system as a hierarchy of layers, with a building block type approach. Sometimes the stacking bears more resemblance to reality than other times, but it is a very simple model to comprehend, and a non-techie can quickly grasp what it's telling them.



Architecture as pasta
Lasagne

Several distinct layers, arranged one atop the other.

Each component is represented by a single block in the stack. The positions in the stack indicate what lives where, how the components relate to each other, and which components can 'see' which other components. Blocks may be placed alongside each other on the same level, and perhaps even can become tall enough to span two layers. To illustrate how this works see the example layer diagram, it shows a system close to my heart: trifle.



A more serious (and famous) example of this is the OSI seven-layer reference model for network communication systems. In all honesty I've worked far more closely with the Goodliffe seven-layer trifle reference model, as has most of the civilised world.

At the lowest level in the stack we find the hardware interface, if our system does indeed interact with physical devices. Otherwise this level is reserved for the most basic service, perhaps the OS or a middleware technology like CORBA. The highest level will likely be occupied by the fancy interface that the user interacts with. As you get further up the stack of layers you move further away from the hardware, happily insulated by the layers in between, in the same way that the roof of a house doesn't have to worry about the magma at the earth's core.

At any point you should be able to brush out all the lower layers and slot in a new implementation of the layer below that honours the same set of interfaces – the system should function as before. This is a key point: it means that you can run the same C++ code on any computing platform that supports your C++ environment. You can swap the hardware platform without touching your application code – relying on the OS layer (for example) to swallow the technical differences for you. Handy.

Being at a higher level means that you can use the public interfaces of the layer directly below. Whether you can use the public interfaces of any lower levels depends of your definition of layering. Sometimes the diagram is fiddled to represent this, as in the sherry brick in the trifle stack. You certainly can't use anything from a higher level; if you break this edict you no longer have a layered architecture, just a pretty diagram drawn in stack form.

Whether components on the same layer can interconnect is again not rigidly defined, but up to the particular definition of layering you choose to adopt.

As you can see, most uses of layering are hardly formal. The relative size and position of boxes gives a clue as to importance of a component, and as an overview that is generally sufficient. The connections are implicit, and in this view the methods of communication irrelevant (however, this can be a key architectural concern for the efficiency of system – you don't send gigabytes of data down an RS232 serial port, after all).

Pipe and filter architecture

This architecture is modelled after the logical flow of data through the system. It is implemented as a string of sequential modules which each read some data in, process it, and spit it out again. Somewhere at the start of the chain is a data generator (maybe a user interface, perhaps

some hardware data harvesting logic), at the end is a data sink (perhaps the computer display). It's Chinese Whispers in digital form. The data flows down the pipe encountering the various filters en route.

The transformations are usually incremental; each filter does a single simple process and tends to have very little internal state. This form is commonly seen in exotic Unix command line incantations, and the pipe and filter architecture is often implemented by this mechanism. If each filter stage requires all its input before spitting out any output, the architecture is essentially equivalent to a batch processing system.

The pipe and filter architecture requires a well defined structuring of data between each filter, and has the implicit overhead of repeatedly encoding the output data for transmission down the pipe and parsing back again in each subsequent filter. For this reason the data stream is usually very simple, just a plain ASCII format, otherwise the burden is too great.

This architecture can make it very easy to add functionality by just plugging in a new filter into the pipeline. However its great downside is error handling. It is hard to determine where an error originated in the pipeline by the time a problem manifests itself at the sink. It's cumbersome to pass error codes down chain towards output stage, it needs extra encoding and is hard work to handle uniformly in several separate code modules. The filters may use a separate error channel (e.g. stderr), but error messages can still easily get mixed up.

Component based architecture

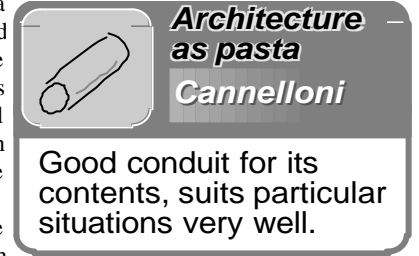
This architecture decentralises the control and splits it into a number of separate collaborating *components*, rather than a single monolithic structure. It is essentially an object-oriented approach, but doesn't necessarily require implementation in an OO language. Each component's public interface is defined in an *Interface Definition Language* (or IDL) which is separate from any implementation.

Component based design arrived with the lure of assembling applications quickly out of prefabricated components, supposedly enabling plug-and-play solutions. It's still up for debate how much of a success this has been. Not all components are designed for reuse (it's hard work), and it's not always easy to find a component for what you want. It's easiest for UIs where popular frameworks and established marketplaces exist.

The core of a component based architecture is a component communication infrastructure, or *middleware*, which allows components to be plugged in easily, to broadcast their existence, and advertise the services they provide. Components are used by looking up this information through some middleware mechanism, rather than by hardwiring a direct connection between two components. Common middleware platforms include CORBA, JavaBeans and COM, each have different pros and cons.

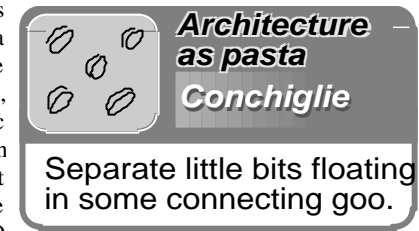
So what is a component? That turns out to be a good question – not everyone agrees on the answer. A component is essentially an implementation unit. It honours one (maybe more) specific published IDL interfaces. This interface is how clients of the component interact with it. There are no back doors. The client is concerned with dealing with an instance of that interface, rather than in how the component is implemented.

Each component is an individual independent piece of code. Behind its interface it implements some logic (perhaps 'business logic' or user interface activity), and contains some data, which may just be local, or may be published (say a file store or database component). Components should not need to know much about each other. If they are tightly coupled then the architecture is just a badly designed monolithic system in disguise as a modern buzzword-compliant product.



Architecture as pasta
Cannelloni

Good conduit for its contents, suits particular situations very well.



Architecture as pasta
Conchiglie

Separate little bits floating in some connecting goo.

[concluded at foot of next page]

Maintaining Context for Exceptions

Rob Hughes <r.d.hughes@open.ac.uk>

This article is based on a concept presented by Andrei Alexandrescu at the 2002 ACCU conference [1]. The basic idea Andrei presented was to store contextual information about the execution stack that can then be accessed in the event of an exception being thrown. In this respect, the idea is very much like using a debugger to display a call stack, except that the information is developer specified context.

Imagine some highly simplistic code that represents the process of building a house; the code might be something like:

```
int main() {
    int some_house_id = ... // Get a house identifier
    BuildHouse(some_house_id);
}

void BuildHouse(int house_id) {
    int num_walls = ... // Establish how many walls
                        // are needed.
    for(int i = 0; i <= num_walls; ++i)
        BuildWall(i);
}

void BuildWall(int wall_id) {
    int bricks_required = ... // Establish number of
                              // bricks required
    for(int i = 0; i <= bricks_required; ++i)
        LayBrick(i);
}

void LayBrick(int brick_id) {
    BrickPos bp = GetBrickPosition(brick_id);
    // lay cement, place brick, point join etc.
}

BrickPos GetBrickPosition(int brick_id) {
    BrickPos bp;
    // calculate the brick position.
    if(bp->Invalid())
        throw InvalidBrickPosException();
}
```

Now if an invalid `brick_id` value gets into `GetBrickPosition()`, it might be reasonable to throw an exception. It is likely that `LayBrick()` doesn't know why the `brick_id` value is invalid, so the exception should be caught further down the call stack. In fact, because the inability to lay a brick means it is difficult to complete the house satisfactorily, it might be desirable to catch the exception in `main()` and then to indicate to the user that the house building process has failed, and in what circumstances it has failed. So the objective of providing contextual information with exceptions is to be able to generate output in a friendly and informative format such as the following:

```
Error: No brick position identified for brick
                                     #brick_id
While getting brick position for brick #brick_id
While laying brick #brick_id
While building wall #wall_id
While building house #house_id
```

Implementation

Ideally, in order to provide the kind of contextual information described above, the code would be rewritten something like the following:

```
int main() {
    // Some code to get a house identifier
    try {
        DO_IN_CONTEXT("While building house")
        BuildHouse(some_house_id);
    }
    catch(const ContextException & e) {
        std::cout << e.what() << std::endl;
    }
}

void BuildHouse(const int house_id) {
    // Code to establish how many walls are needed.
    for(int i = 1; i <= num_walls; ++i) {
        DO_IN_CONTEXT("While building wall")
        BuildWall(i);
    }
}
```

The component based architecture may be deployed in a networked environment with components on different machines, but can as easily be a single machine installation. This may depend on the type of middleware in use.

Frameworks

Instead of developing a new architecture for a specific project it may be appropriate to use an existing *application framework* and slot development into that skeleton. A framework is an extensible library of code (usually a set of co-operating classes) that forms a reusable design solution for a particular problem domain. When using a framework most of the effort has been done for you, with the remaining pieces following a fill-in-the-blanks approach. Different frameworks will follow different architectural models – by using a framework you commit to its particular style.

Frameworks differ from traditional libraries in the way they interact with your code. When using a library you make explicit calls into the library components under your own thread of control. A framework turns this around; it is itself responsible for the structure and flow of control. It will call into your supplied code as and when necessary.

Most of the popular frameworks available are for the user interface domain.

Alongside the use of off-the-shelf frameworks is the consideration of design patterns. Whilst not an architectural style in their own right, patterns

are small-scale architectural templates. Usually employed at the design level rather than in the system architecture, they are micro-architectures for a few collaborating components, distilling a recurring structure of communication. A design pattern solves a general design problem within a particular context. Patterns are a set of design best practices, and are described in the ubiquitous GoF book [3] and numerous subsequent publications.

Conclusion

The Roman architect Vitruvius made a timeless statement of what constituted good design: strength (*firmitas*), utility (*utilitas*), and beauty (*venustas*) [4]. This holds true for our software architectures. Without a well-defined, well communicated architecture, a software project will lack a cohesive internal structure. It will become brittle, unstable and ugly. Eventually it will reach a breaking point.

All this talk of pasta has made me hungry. I'm off to build a seven-layer reference trifle.

Pete Goodliffe

References

- [1] Barry Boehm. *Software Engineering Economics*. Prentice Hall, 1981. ISBN: 0138221227.
- [2] Frederick P. Brooks, Jr. *The Mythical Man-Month, Anniversary Edition*. Addison Wesley, 1995. ISBN: 0-201-83595-9.
- [3] Erich Gamma, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1997. ISBN: 0-201-63361-2
- [4] Marcus Vitruvius Pollio (c. 70-25 BC). *De Architectura*. Book 1, Chapter 3, Section 2.



```

void BuildWall(const int wall_id) {
    // Code to establish number of bricks required
    for(int i = 1; i <= bricks_required; ++i) {
        DO_IN_CONTEXT("While laying brick")
        LayBrick(i);
    }
}
// etc.

```

At first appearances, it might be thought that `DO_IN_CONTEXT` is a macro concealing a try-catch block, which catches the propagating exception, adds its contextual information to the exception message, and then throws a new exception. One of the main arguments Andrei put forward was that this would be an unnecessary complication; instead it is preferable that the macro creates an automatic object whose destructor does the necessary work during stack unwinding. The idea was to make use of `std::uncaught_exception()` in the destructor of this automatic context object to determine if an exception is propagating, and therefore whether to add the context information it contains to the exception.

After the conference, Alisdair Meredith started a thread on `comp.lang.c++.moderated` discussing this concept [2]. The upshot of this discourse was that the original idea was flawed. Conceptually, the context object might have looked something like the following:

```

class Context {
public:
    Context(const std::string & context)
        : ex_at_start_(std::uncaught_exception()),
          context_(context) {}

    ~Context() {
        if(std::uncaught_exception() && !ex_at_start_) {
            // An exception occurred after the constructor
            // executed and before the destructor.
            // Add the context information held to the
            // exception
        }
    }

private:
    bool ex_at_start_;
    std::string context_;
};

```

The problem as discussed on `c.l.c.m` is that there is no way to access the propagating exception without catching it; precisely the situation this object was designed to avoid. The solution Andrei proposed is to create a context stack instead of adding the contextual information directly to the exception. This could either be some kind of static object in a single-thread application, or thread local storage in a multi-threaded application.

This mechanism can still make use of an automatic context object. Instead of storing the context information locally in the object, the context object's constructor pushes it onto the context stack. Then the context object pops the last context item from the stack if its destructor is reached without an exception being thrown. If an exception is detected in the destructor, the context stack is left unchanged and the whole of the context information can be retrieved from the stack at the point where the exception is handled. A basic implementation of this idea for single-threaded usage is given below.

```

// rhex_static.hpp

#ifndef RHEX_STATIC_HPP
#define RHEX_STATIC_HPP

#include <string>
#include <deque>
#include <ostream>

#define DO_IN_CONTEXT(str) \
    rhex::Context context(str);

```

```

#define DO_IN_SCOPED_CONTEXT(str) \
    { rhex::Context context(str);
#define END_SCOPED_CONTEXT }

// rhex::ExContextHolder

namespace rhex {
class ExContextHolder {
public:    // Queries
    static ExContextHolder& Holder();
    std::string LastContext() const;
    void DumpContexts(std::ostream& dest);
    // Modifiers
    void AddContext(const std::string& context);
    void PopLastContext();
private: // Constructors and destructors
    ExContextHolder() {};
    ExContextHolder(const ExContextHolder& rhs);
    ExContextHolder& operator=
        (const ExContextHolder& rhs);
    // Data members
    std::deque<std::string> contexts_;
}; }

inline
rhex::ExContextHolder&
rhex::ExContextHolder::Holder() {
    static ExContextHolder context_holder;
    return (context_holder);
}

inline
std::string
rhex::ExContextHolder::LastContext() const {
    return (contexts_.empty() ? std::string() :
    contexts_.back());
}

inline
void
rhex::ExContextHolder::DumpContexts
    (std::ostream& dest) {
    while(!contexts_.empty()) {
        dest << contexts_.back() << std::endl;
        contexts_.pop_back();
    }
}

inline
void
rhex::ExContextHolder::AddContext
    (const std::string& context) {
    contexts_.push_back(context);
}

inline
void
rhex::ExContextHolder::PopLastContext() {
    if(!contexts_.empty()) {
        contexts_.pop_back();
    }
}

// rhex::Context

namespace rhex {
class Context {
public:    // Constructors and destructors
    explicit Context(const std::string& context);
    ~Context();
private: // Data
    bool exception_present_;
}; }

```

```

inline
rhex::Context::Context(const std::string& context)
: exception_present_(std::uncaught_exception()) {
if(!exception_present_) {
rhex::ExContextHolder::Holder().AddContext
(context);
}
}

inline
rhex::Context::~~Context() {
if(!std::uncaught_exception()
&& !exception_present_) {
rhex::ExContextHolder::Holder().
PopLastContext();
}
}

#endif // RHEX_STATIC_HPP

```

A `std::deque` object is preferred as the stack container because it offers the opportunity to add more extensive access features to the `ExContextHolder` instance should it be required. A `std::stack` container adapter could be used, but as the default container for this adapter is a `std::deque`, there is no real benefit to doing so in this case.

In most cases the `DO_IN_CONTEXT()` macro is sufficient. In some cases it may be necessary to use the pair of macros `DO_IN_SCOPED_CONTEXT()` and `END_SCOPED_CONTEXT()` (for users of Borland C++ Builder, a slightly modified version is required due to an apparent library bug¹). These allow the automatic `Context` object to be wrapped in a scope, allowing for nested contexts within a single function, although in general needing to do this probably indicates a design flaw as it implies the function is doing too much work. It allows for constructs such as:

```

void foo() {
DO_IN_SCOPED_CONTEXT( "While in outer scope" )
// Some processing which might produce an
// exception
DO_IN_SCOPED_CONTEXT( "While in inner scope" )
// Some more processing which might produce an
// exception
END_SCOPED_CONTEXT
END_SCOPED_CONTEXT
}

```

which might produce output such as:

```

An error occurred
While in inner scope
While in outer scope

```

Discussion

As an interesting aside, the use of the Singleton pattern in this work was an issue of some considerable concern to the author. This particular use appears to be an extremely rare case where use of a singleton can be justified although is not completely necessary. The code presented here is a work-around required because of an ineffective language feature [3], `std::uncaught_exception()`, and so is designed to be much like a language feature; easy to use and non-intrusive.

So why use a Singleton? It might, for example, be preferable to force the client to create an instance of the context stack at the beginning of each

critical execution path, which is then passed down the call chain as an additional parameter. But this imposes a clumsiness and additional overhead for the client programmer. Now users have to decide when to create context stacks, where to pass them as parameters and where they should be part of the state of important classes. Furthermore, context stacks may frequently need to be passed to functions that don't actually need them, so that they can be passed on down the call chain. Experience shows that if this kind of feature is hard to use, developers just won't bother. For single threaded applications, a Singleton is a simple, non-intrusive and easy to use solution.

There is a question mark over how useful providing this sort of context information is. It can be argued that this provides very little useful information to the end-user, and nothing that a developer can't get from other sources. In this case, it would be difficult to argue that the extra overhead involved was worthwhile. In complex applications, however, where exceptions may be handled a long way from their source, the extra information is invaluable. Even with the simple house building example presented earlier, it is easy to see the benefit. Imagine if this formed part of a larger system, and houses were not the only type of building that could be constructed with walls. The bricklaying functions know nothing about what structure the bricks are for, so the exception either has to have no information about the type of structure, or coupling needs to be increased by passing information up the execution stack. If this formed, for example, part of an automatic batch job or similar, which would be a preferable message for the operator upon their return: one saying 'Invalid brick position' or one saying 'Invalid brick position while building house X'? Picture also the bug report that your customer files; which message attached to the report will be easier for the maintainer to get to grips with?

For many applications, where the additional overhead imposed is acceptable, the benefits will be considerable, either through better information, or through a better design because of reduced coupling between functions, or both. The author's own experience of using this mechanism on small applications has already shown benefits. The very first unintentionally generated exception encountered when using this code immediately showed that some unexpected recursion was happening, and saved considerable time in hunting down the source of the problem.

The implementation presented here is very simple, and there is clearly scope for a much more advanced system for large applications. One idea that Andrei has suggested [4] would be to store a generic base class in the context stack rather than strings. The base class would specify some kind of simple output interface, similar, for example, to the standard `printString` method in all SmallTalk classes. Developers would then create more complex, application specific context classes derived from this base class, which know how to display their context information.

Where to use such context providers is also an interesting question. The ideal situation would be to only deploy context objects along execution paths that could result in an exception, and where the additional context provided will be useful. In reality, the complex nature of most software makes it difficult, if impossible, to identify all possible exceptional execution paths [5]. Instead the emphasis must be on using context objects at key points; where either significant runtime information can be added to the context stack, or along the principal execution paths for the software.

Rob Hughes

Acknowledgements

Thanks to Pete Goodliffe for encouraging me to submit this article and for providing a helpful informal review of it.

References

- [1] Alexandrescu, A. 2002. Error Handling in C++. *ACCU Spring Conference 2002*
- [2] <http://groups.google.com: thread start id: 3CB69B73.9E8BA09C@uk.renaultf1.com>
- [3] Sutter, H. 2002. *More Exceptional C++: 40 New Engineering Puzzles, Programming Problems, and Solutions* {Item 19}. Addison-Wesley.
- [4] [http://groups.google.com: article id: a9913b\\$18o9e\\$1@ID-14036.news.dfncis.de](http://groups.google.com: article id: a9913b$18o9e$1@ID-14036.news.dfncis.de)
- [5] Sutter, H. 2000. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions* {Item 18}. Addison-Wesley.

1 Rather annoyingly, `std::uncaught_exception()` doesn't seem to work properly with Borland C++ Builder. I found that it always returned `false` even when an exception was propagating. The upshot was that the contexts that should have been left on the context stack were popped when the various automatic `Context` objects were destroyed during stack unwinding. The slightly ugly solution to this is to push and pop the context strings manually, avoiding the use of `Context` at all. This requires a pair of macros for both scoped contexts, but obviates the need for unscoped contexts, e.g.,

```

#define DO_IN_CONTEXT( str )\
rhex::ExContextHolder::Holder().AddContext( str );
#define END_CONTEXT\
rhex::ExContextHolder::Holder().PopLastContext();

```

When Worlds Collide

#1 – Embedded Systems and General Purpose Computers

Mark Easterbrook

Until recently the worlds of Embedded Systems and General Purpose Computers were two distinct branches of the computer/electronics industry, each with their own requirements, tool sets, philosophies, applications, markets and customers. First we must look at them individually before we can understand the issues when they meet.

Embedded Systems

The term Embedded Systems is often used synonymously with Real Time Systems, but the two terms are not the same. Real Time has a distinct definition; the system must perform within given time constraints, whereas Embedded System is a general term for a non-general purpose computer embedded in another device. With no explicit definition, we can only identify Embedded Systems by the characteristics they are likely to have:

- No or limited human interface. A vehicle engine management system only interfaces to the mechanical and electronic parts of the car, whereas a washing machine controller has a limited human interface through simple buttons and display.
- Provides only one or a set of closely related functions. A D-A converter in a CD player provides one clearly defined function, whereas a IP router performs many functions but all related to the management and routing of IP packets.
- Contains program code in solid-state storage such as PROM that cannot be user upgradeable. The code for a mobile phone is often fixed for the life of the product but that of the telecom switch carrying a call from that phone will be upgradeable, usually without taking the switch out of service.
- Either has no operating system or uses a small footprint embedded or real-time OS (RTOS).
- All code is usually from one source (usually in-house) maybe with some bought-in libraries such as communications stacks. This means that all code running on the device is under the same control and thus can be fully integration tested before shipping and simple scheduling algorithms such as round-robin co-operative multi-tasking are safe to use.
- The code is often designed and implemented for one proprietary hardware platform. The hardware and software are considered a single unit; they are useless on their own.
- The memory and CPU power is usually constrained. On small devices manufactured in large quantities a few pence saved on a cheaper processor or smaller memory produces savings that dwarf the extra software development effort to fit the software to the limited hardware. Larger systems may be limited by having to use parts that will remain available for the product's predicted lifetime rather than using faster state-of-the-art components which may be obsolete quickly.
- The product is defined by what it does, the how is largely irrelevant. A customer buys a washing machine with digital control, a fuel-injected engine, or a radio with digital tuning. They don't buy a computer that does washing, runs an engine, or tunes to radio stations.

General Purpose Computers

The PC has evolved from its original intention as a "Personal" general-purpose computer to something so powerful it is practically the "ultimate" General Purpose computer. Many functions such as speech recognition and real-time video and audio encoding and decoding no longer need specialised platforms and the "humble" PC now has the power to perform these. Even super-computers are under threat from clusters of relatively cheap PCs. As with Embedded Systems, it is better to characterise General Purpose Computers rather than define them:

- Thought of as a computer first, and a solution second. It is a computer running a word processor, rather than a word processing machine, a computer performing weather prediction rather than a weather predictor.
- Even if dedicated to only one task, it is equally capable of doing lots of other tasks. The choice to dedicate to one purpose is purely an operational one and not to do with the underlying machine.
- Has a rich human interface. Even the DOS command line is extensive compared with the interface to a digital alarm clock.

- Has an operating system that allows multiple users and multiple applications to run so that one user or application can be prevented from interfering with the others.
- Can run applications or even operating systems from different suppliers. It is common for the hardware and software to be produced by different companies, or different divisions within a large company.
- Any specialisation is general rather than specific. The system may excel at floating-point arithmetic, parallel processing, or database storage, but many applications may exist that make use of these traits. A system such as a Cray super-computer may be viewed as rather specialist, but it really is a general-purpose computer compared with, say, a nuclear power station controller or telecommunications switch.
- Are often highly configurable. A mainframe may be available in many different sizes (scalable) whereas the PC has a seemingly infinite number of combinations of processors, buses, cards, etc.
- Are user-upgradeable, especially in terms of software.

Collision

As available processing power increases, size reduces, and prices drop, there is an increasing grey area where it is difficult to decide if a particular device is an embedded system, or a general-purpose computer. Embedded Systems is no longer a domain of 4- and 8-bit microprocessors, but increasingly of 32- or even 64-bit, often using the same CPU families chosen by the designers of General Purpose computers, because these are now powerful enough to perform tasks in software that previously required custom hardware-software solutions. Embedded Systems are increasingly able to perform tasks other than their core intent because they share base components with more general purpose platforms, and General Purpose computers are increasingly being used where an Embedded System would traditionally have been chosen because they [General Purpose computers] are now a powerful and cheap enough alternative.

Nowhere is this collision more apparent than the PDA (Personal Digital Assistant). It often is based on custom hardware typical of embedded systems with limited processing power and memory, and upgrade opportunity, contains its core program in ROM memory and has software written only for the one hardware platform. Yet it runs a number of applications such a word processing and web browsing, interacts directly to a human operator through a visual user interface, can be easily upgraded with replacement or new applications, and is often considered a computer with applications rather than a diary that happens to be electronic. So is a PDA an embedded system, or a general-purpose computer? Is it a very flexible embedded system, or a lightweight laptop PC?

Operating Systems

In the arena of Operating Systems – the glue that binds together the applications – the jury is still out. On one side, existing embedded OSs and new OSs based on embedded OS theory such as EPOS are being pushed by their respective supporters, and on the other, cut down versions of general purpose computer OSs such as Windows CE and Embedded Linux are being developed.

The Embedded Programmer

The embedded programmer is used to working within a tight budget of processing power and memory, and often using only simple 8-bit processors. In comparison, a modern 32-bit PDA has plenty of power and memory and should pose no challenge. However, there is one part of the PDA which will be alien to most embedded programmers: the graphical user interface (GUI). With the GUI come a number of problems that cannot be solved using traditional embedded methods: memory management, task management, error handling, etc.

Many programmers choose to work in the embedded world because the rules are clearly defined, albeit sometimes difficult. Writing hardware interface code requires care and precision, an understanding of how hardware works, and logic, but it is well defined – it works in a logical manner and the error conditions are usually documented. The usual answer to memory and task management in an embedded system is to allocate everything at start up – this is easy because the tasks and their memory requirements are known at start up, they may even be known at design time, therefore error conditions relating to these don't have to be handled during normal running. In fact, most resources can be handled in this way, for example, a serial port will only be used by the task designed to use it so allocation and de-allocation and the resultant resource locking is not required.

[concluded at foot of next page]

Mixing Strings in C++

Sven Rosvall <sven-e@lysator.liu.se>

Every (sane) programming language has some mechanism to handle text strings. Text strings are basically a sequence of characters. But this sequence can be implemented in different ways, which are not always interchangeable in a simple and efficient way.

This article is not going into the realm of how characters are represented and conversions between character representations, instead the interaction between different string implementations are investigated.

Strings in C

There is only one string type in C. This makes life easy as there is no confusion in the choice of string type to use. C uses a linear sequence of characters terminated by a null character. This sequence is either kept in an array or some other memory pointed to by a pointer. An array of characters is easily converted to a pointer to characters and this makes it easy to pass around pointers to characters in function parameters and data structures. The pointers are so often used to represent strings that they often are referred to as strings and not pointers to strings.

Working with these C strings is not always easy, as the memory for the character sequence must be managed by the programmer. To append a string to another you must reserve a piece of memory to store the new string, then copy the original string and append the second string to this. Then you can use a pointer to this new memory as the result string. Just don't forget to free the memory of the original string to avoid memory leakage, but not if anyone else is still using a pointer to it! This handling gets even worse when you need to merge several strings, for example when adding a file name to a directory name:

```
char * dirname = ...;
char * filename = ...;
char * tmpdirname;

// Length of new string includes directory
// separator and null character
int newlen = strlen(dirname) +
            strlen(filename) + 2;
tmpdirname = (char *) malloc(newlen);
if (tmpdirname == NULL) {
    // Handle memory error.
}
strcpy(tmpdirname, dirname);
strcat(tmpdirname, "/" );
```

So, the new embedded programmer has to learn how to operate in the brave new world of user interfaces. A user of a PDA will be presented with a graphical interface much closer to a general purpose computer than any of the embedded devices he has currently come across, therefore he will expect them to behave as such. New programs can be loaded and executed at any time, which means resource allocation and contention need to be considered. There will probably be some PC card or removable memory slots, so resources may appear and disappear.

The Application Programmer

A typical application programmer has been fairly spoiled in recent years. As well as a large choice of development environments, the speed of processors and size of memory have increased at a fast rate. Rarely does the programmer of a modern day general-purpose computer have to spend time tuning code for space or performance, and when these become an issue such as in servers handling a high number of clients, it is design time decisions rather than programming idioms that yield the greatest results.

Sometimes the programmer does not even need to address the problem, as it goes away by itself due to the fast pace of change in the industry: A program that is sluggish during development time is acceptable at the point of release, and fast once shipped in numbers as hardware speeds outpace software development. Users have become used to having to upgrade their hardware to run the latest software so they almost expect it, so new features can eat up memory and processor cycles without fear.

The computer industry, believing there is a shortage of programmers, has accepted into its ranks scores of developers who otherwise would not

```
strcat(tmpdirname, filename);
free(dirname);
dirname = tmpdirname;
```

Of course, most developers wrap such functionality in utility libraries. However, the amount of variations of semantics makes such libraries difficult to use. In this example, we append a filename to a directory name that involves a third string temporarily. Yes, a simple function to append a string to another could be used within this function but then there would be two calls to that function and thus two memory allocations. So two functions are needed, one for simple string append and one for file name append. For every special semantic variation needed, the number of functions increases.

What if the directory name referred to by `dirname` cannot be freed? It might be a pointer to a string literal or a character sequence stored in an array on the stack and cannot be freed. Or the string is still used somewhere else in the program. We need to have two variations each of the functions in the library, one that frees the original string and one that leaves it intact. Sometimes the variants that free memory are left out and the user of the functions has to remember to free the memory.

Strings in C++

C++ provides the capability to create a string class that manages the underlying memory and makes strings easier to use. Standard C++ [1] introduced a string class in the library to encapsulate the semantics of strings. Appending a file name to a directory name using the C++ string type is much simpler:

```
std::string dirname = ...;
std::string filename = ...;

dirname = dirname + "/" + filename;
```

Note that the error handling is done with exceptions here. This may look like cheating. But exceptions are intended to let code avoid error handling in the cases where nothing can be done other than pass the error on to a higher level.

So using a string class makes life easier for the programmer. Sadly, this is not the only way strings are used in C++. C++ has inherited much from C, including its string concept. Many libraries use C strings (pointers) for function parameters and data structures so both C and C++ programs can use them. One such library many C++ programmers use is the C library. C strings are also used in some parts of the C++ library where exceptions cannot be thrown.

make the grade. This has led to many companies where sloppy work has become the norm. The tolerance of the users to having to restart the program or machine, and of operating systems that tidy up the resources, means such practices proliferate.

These factors have led to the explosion of applications giving the user unprecedented choice. In recent years the exponential growth in open source programs has added to the existing commercial and shareware offerings increasing this choice further.

Yet, the most common "computer" on the planet is still embedded.

The Outlook

For a minority at the extremes, such as the programmer of 8-bit controllers and the database query language programmer, the colliding of worlds is irrelevant. For the rest of the computer industry many facets of the job will change, desktop applications will require a PDA version, server code will need to deal with a larger variety of clients, embedded systems will require graphical user interfaces and allow user driven upgrades. The application programmer will not only need to learn different platforms, but will be required to learn some of the skills the embedded programmer takes for granted, conversely, the embedded programmer will need to deal with interfaces that are no longer well defined or predictable and will no longer program in an environment mostly under his control.

The future will certainly require many of us to learn new skills, but our industry has always been in a state of rapid change. The better developers will expand their horizons and meet the challenge; the least skilled will struggle. As they say in France, "Plus ça change, plus c'est la même chose".

Mark Easterbrook

Conversions between String Types

C++ strings are very convenient to use. C strings are relatively easy to use if ownership of the memory is managed properly. However, mixing them can cause a few headaches.

C strings from C++ strings

When you have a C++ string and want to pass it to a function that takes a C string, you can easily get a `const char*` using the `c_str()` member of `std::string`.

```
// A function that promises not to modify
// the string.
extern void f(const char *);
std::string myString = ...;

f(myString.c_str());
```

However, the pointer you get points to some data that is managed by the C++ string object. This data will only be valid as long as the string object is not modified in any way. If `f()` only uses the pointer in its body to do some processing or copying then everything will be fine. But, if `f()` stores the pointer in some data structure that lives on after `f()` has returned there is no guarantee that that pointer points to valid data when it is looked at again. So keep your tongue nicely in your mouth and all will be fine. Right?

Well, there are some more situations you have to watch out for. What if `myString` is modified by another thread while `f()` is still executing? The behaviour will differ depending on your library implementation and how the string is modified. In some situations, you may get away with this because the memory for where the C string is stored is not overwritten immediately. You may end up with a debugging nightmare instead where your string looks OK for a while and then suddenly changes contents or gets corrupt.

You are not safe in a single-threaded system either. What if `f()` calls some other function that modifies the same string object that was used in the call to `f()`?

C++ strings from C strings

Going from C strings to C++ strings is not dangerous but there are still a few things to keep in mind for efficiency.

The `std::string` class has a convenient conversion from C strings that can be used for implicit conversions from pointer to characters.

```
// A function that promises not to modify
// the string.
void g(std::string const &);
char* myString = ...;

g(myString);
```

The call is clean here, no trickery is required to call `g()`. However, you must be aware that a temporary `std::string` object is created implicitly. This object keeps a copy of the character sequence allocated on the heap and there are invisible calls to the constructor and the destructor of `std::string`. In some projects, this cost cannot be afforded.

A simple way to avoid this temporary object is to provide an overloaded `g(const char*)` at the cost of a second function body.

Custom String Classes

To make things worse, there are several custom string class implementations around. Some predate the C++ standard and some add various features needed in their projects. These non-standard string classes usually work in a similar way to `std::string` but have some minor additions or omissions that make them difficult to replace with `std::string`.

My most recent project uses two custom string types. The first is a home-made string dating from the days before the C++ standard that also adds a hash value for the string (called `xstring`). This hash value was used in various tables to find the string quickly.

The other custom string type is a convenience class, a wrapper for `std::string` (called `U_String`). It is derived from `std::string`

and thus behaves exactly like it, but can be forward declared without any `#include`. This is very useful for header files that only declare string parameters passed by const-reference. (Where the type is only used by name.) The header file `<string>` includes `<iostream>` in some libraries, which can be very heavy for the compiler, especially when pre-compiled headers cannot be used. Compile times were reduced significantly when the `U_String` class was introduced because the project consisted of a large number of small translation units.

Older parts of the project use the `xstring` class while newer parts use `U_String`. Converting between them is trivial to code but requires creation of a temporary object. Usage of the `xstring` class is slowly being replaced by `U_String` where the hash value isn't used. The intent is to remove the `xstring` class completely. The need for the hash value will be replaced by a flyweight string (see below).

String Semantics

The C++ string is a powerful tool. It does a very good job to represent a string. However, there are some problems with it for power-users.

In most library implementations, each string object keeps a copy of the character sequence in a piece of memory it owns. This may be expensive if the same string is copied many times.

Some implementations have solved this by sharing exact copies between copied string objects by keeping a reference count for the memory that contains the character sequence. The memory is shared until a string object is modified. It then gets a bit of memory for itself. This is called COW-strings. (Copy-On-Write) This technique reduces the memory consumption drastically in some situations. It also cuts down the time for allocating new memory on the heap and copying the character sequence. However, this technique may cause problems in threaded systems and it is not used in most modern library implementations.

My recent project is a C++ parser that tokenises its input. Each token contains a file name, line and column numbers. Of course, there are many tokens from the same file. The project uses two different compilers with different library implementations. The memory consumption differed dramatically between the two implementations. Naturally, we wanted the best performance with both implementations.

Read-only strings

The first approach was to create a read-only string class. File names used in tokens are not likely to change. So a reference counting technique could be used. We had seen the benefit of this in one of the library implementations. If we removed all modification methods of the string class, we would eliminate the logic required to keep the read-only strings consistent. To make the implementation simple we just created a class that contained a reference counted pointer to `std::string` and added the methods we required.

One benefit of this read-only class we saw immediately was that when two filenames were compared, we could compare the pointers first and if they were equal, there was no need to compare each character in the strings.

Flyweight (read-only) strings

The thought of using pointer comparisons for string comparisons was very appealing. However the read-only strings could be created from different sources and thus there would exist equivalent strings that did not have equal pointers.

The next step was to use the Flyweight Pattern, see [2]. Each file name was now represented by a handle. File name strings were created by a factory to guarantee that equivalent strings used the same handle.

The Flyweight string class used policy classes that specify how the strings are stored and a usage domain. The domain was introduced so that different kinds of strings couldn't be compared. Separating strings in domains also keeps the numbers of strings in each domain down, therefore the time for inserting the string is lower. We had one domain for filenames, another for identifiers etc. The compiler would complain if we tried to compare filenames and identifiers.

The storage parameter specifies the internal data structure for the strings and what type the handle is. The initial implementation used a set of strings for storage and the handles were iterators into the set.

Conversions Again

Now we suddenly have a project with six different types of strings. (More if each specialisation of the specialised flyweight strings are counted.) Care

[concluded at foot of next page]

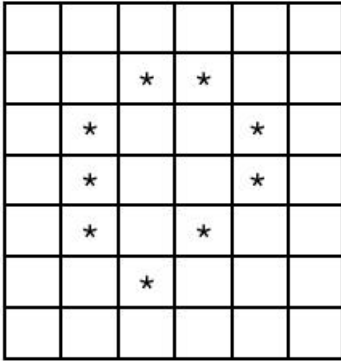
A Polygon Seed Fill Algorithm

James Holland <jamie_holland@lineone.net>

It was interesting to read in C Vu 15.2 that Francis would be interested to hear from anyone who knows of a good fill algorithm. I recalled having a go at getting a seed fill algorithm to work several years ago. In those days I was writing in Pascal so I thought it would be an interesting exercise to rewrite it in C++.

The first task was to find the source of the algorithm. After a bit of a search I found a description of the algorithm by Rogers [1]. Unfortunately, his description of the algorithm is not all that clear and the pseudo-code provided would never have worked as far as I could see. I had to do a lot of reading between the lines before I thought I understood the algorithm the author was trying to describe. I remembered having this trouble the first time around.

The algorithm fills an area bounded by either the screen limits or by pixels of a specified colour (the bounding pixels). The bounding pixels must form a continuous 'fence' otherwise the fill colour will leak out through any gaps. A border is continuous if it is impossible to move from an interior pixel to an exterior pixel by only horizontal or vertical steps without passing through a border pixel. The diagram below shows a completely continuous border and may help to visualise the situation.



The algorithm attempts to fill the enclosed region by starting from an arbitrarily chosen 'seed' pixel. The seed pixel can be either inside the boundary or it can be outside the boundary. Taking the above diagram as an example, if the seed is inside the boundary, then the central region will be filled. If the seed is outside the boundary, then the area outside the boundary will be filled.

The algorithm works as follows.

The seed pixel location is pushed onto a stack and the algorithm's main loop is entered. The loop will exit at this point only when the stack is empty. The body of the loop immediately pops the pixel location from the stack. The pixel at that location is filled. The algorithm then fills the row of pixels to the left of the seed until either a boundary pixel is found or the limit of the screen is reached. The horizontal position of the last pixel to be filled is stored and represents the extreme left of the fill line. Next, the pixels on the row to the right of the seed are filled until either a boundary pixel or the screen limit is reached. The location of the last pixel to be filled is stored and represents the extreme right of the fill line.

The algorithm now considers the row above the seed pixel (if that row exists). This line is scanned from the previously stored extreme right position to the extreme left position, inclusively. No pixels are filled during this upper scan; instead the first non-border pixel location is pushed onto the stack. If this upper scan line is broken by groups of border pixels, the

first non-border pixel between each group (in the direction of the scan) is also pushed onto the stack.

The row below the seed pixel is now scanned in exactly the same way as the row above the seed pixel and pixel locations pushed onto the stack accordingly.

This completes the first iteration through the main loop. The stack is checked to see whether it is empty and, assuming it is not, the last pixel location to be pushed on the stack is popped off (to form the next seed, in effect). The filling and scanning process is repeated. Eventually, the stack is exhausted and the algorithm terminates. All bounded pixels have been filled.

As an example, consider the 10 x 10 display grid below. The border pixels are shown as asterisks and the original seed (at location 6, 5) is marked by an 'S'. The numbers within the area to be filled show the order and location of pixels pushed onto the stack during the fill process. The shaded background indicates the position of the pixels to be filled.

	0	1	2	3	4	5	6	7	8	9
0		*	*	*	*					
1	*		9			*	*			
2	*	8		*	*		11	*		
3	*	7	*			*	10	*		
4	*	2	*	*	*			1	*	
5	*						S		*	
6		*	4	*	*	*		3		*
7		*	5	*		*		6	*	
8		*	*				*	*		

First, the seed pixel is filled. The five pixels to the left of the seed are filled and then the single pixel to the right of the seed is filled. The extreme left and extreme right scan limits are stored as 1 and 7 respectively. The row above the seed pixel is now examined within the scan limits. It can be seen that row 4 is divided into two groups of unfilled pixels. Accordingly, the locations at 7, 4 and 1, 4 are pushed onto the stack. Similarly, the row below the seed pixel is examined within the scan limits. This row is also found to be divided into two groups and the pixels at locations 7, 6 and 2, 6 are pushed onto the stack. This completes one pass of the algorithm. The stack is not empty and so the pixel location 2, 6 (the last one to be pushed) is popped off the stack. The process continues by filling the new seed pixel at location 2, 6. The pixel at 2,7 is now pushed onto the stack and so on. Eventually, all the bounded pixels are filled, there are no more pixels to push onto the stack, the stack becomes empty and the process ends.

The theoretical basis for this algorithm, as Rogers indicates, is described in a paper by Shani [2].

My C++ realisation of the seed fill algorithm is as direct as possible. The only C++ feature of note is the use of the standard template library `stack` class. The algorithm could just as well be written in any language that supports the concept of a stack. A stack could also be constructed by hand and thus make practically any programming language suitable.

A 20 x 20 display grid is included in the example code for the purposes of testing. An arbitrary border has been defined that includes an inner 'hole'.

had to be taken when there was conversion between them. However, it turned out that having separate types for each kind of string was more helpful than confusing. Having separate types gave the compiler a chance to help us in finding inconsistent usages of string types. There wasn't that much conversion between the new string types and ordinary strings so the small cost was acceptable.

Conclusion

Strings are used for many purposes. Although a string is a simple concept, there are many ways to use strings, and many types of strings tailored for these usages. You may have many different string types in your project

depending on how the strings are used and on the history of your project. If your project uses strings heavily, you may optimise your code by looking at how different string types are used and interact with each other and decide on which type of string is best suited for each usage.

Sven Rosvall

References

- [1] *International Standard: Programming Languages – C++ ISO/IEC 14882:1998(E)*, 1998.
- [2] Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

```

#include <stack>
#include <iostream>

const int width = 20;
const int height = 20;

char screen[height][width] =
{"          ***** ",
 "    ***   *       * ",
 "  *   * *   ***** ",
 " *     ***   *       ",
 " *           *       ",
 " *     *****   *   ",
 " *   * * *   * *     ",
 " * *         ***** ",
 " * *           ",
 " *   **      ***** ",
 " *     ****      *   ",
 " *           *       ",
 " *     *****   *   ",
 " * *   **      *     ",
 " * * *   *   ***** ",
 " *   *** *   *       ",
 " *     **   ***      ",
 " *****          *   ",
 "          *****   "};

struct Location {
    int column;
    int row;
};

std::stack<Location> seed_stack;

void display() {
    // Display the image.
    for (int row = 0; row < height; ++row) {
        for (int column = 0;
            column < width;
            ++column) {
            std::cout << screen[row][column];
        }
        std::cout << std::endl;
    }
    std::cout << std::endl;
}

void pixel(const Location & position,
           char character) {
    // Place the character on the screen
    // at the specified location.
    screen[position.row][position.column]
        = character;
}

char pixel(const Location & position) {
    // Get the pixel at the specified location.
    return
        screen[position.row][position.column];
}

int main() {
    display();

    Location seed_location = {7, 4};
    seed_stack.push(seed_location);

    while (!seed_stack.empty()) {
        Location location = seed_stack.top();

        // Fill the pixel at the seed location.
        pixel(location, '*');

```

```

// Fill the line to the left of the seed.
--location.column;
while (location.column >= 0
      && pixel(location) != '*') {
    pixel(location, '*');
    --location.column;
}
int extreme_left = location.column + 1;

location.column
    = seed_stack.top().column + 1;
seed_stack.pop();

// Fill the line to the right of the seed.
while (location.column < width
      && pixel(location) != '*') {
    pixel(location, '*');
    ++location.column;
}
int extreme_right = location.column - 1;

// Scan above the seed row.
--location.row;
if (location.row >= 0) {
    bool previous_pixel_is_border = true;
    for (location.column = extreme_right;
        location.column >= extreme_left;
        --location.column) {
        if (previous_pixel_is_border
            && pixel(location) != '*') {
            seed_stack.push(location);
            previous_pixel_is_border = false;
        }
        else
            if (pixel(location) == '*') {
                previous_pixel_is_border = true;
            }
    }
}

// Scan below the seed row.
if (location.row < height - 2) {
    location.row += 2;
    bool previous_pixel_is_border = true;
    for (location.column = extreme_right;
        location.column >= extreme_left;
        --location.column) {
        if (previous_pixel_is_border
            && pixel(location) != '*') {
            seed_stack.push(location);
            previous_pixel_is_border = false;
        }
        else
            if (pixel(location) == '*') {
                previous_pixel_is_border = true;
            }
    }
}
}
display();

return 0;
}

```

James Holland

References

- [1] Rogers, David F. (1988) *Procedural Elements for Computer Graphics*, McGraw-Hill. ISBN 0-07-Y66503-6
- [2] Shani, Uri "Filling Regions in Binary Raster Images: A Graph-Theoretic Approach", *Computer Graphics, Vol. 14*, pp. 321-327, 1980 (Proc. SIGGRAPH 80).

Reviews

Bookcase

Collated by Michael Minihane
<michaelm@pobox.co.uk>

Francis Glassborow writes:

You may have heard that The PC Bookshop Ltd is no more. However its important component parts (the shop in Sicilian Avenue and the website) were acquired by Holborn Books Ltd – which is owned by James Lake who founded the PC Bookshop a decade ago.

Books published by O'Reilly & Associates seem to dominate the reviews in this issue. Not all of them are for good books. It seems to me that this publisher is currently generating too many new titles with the result that not all of them are worth shelf space. I think that publishers as well as authors can be guilty of 'pot boiling'. In the long run such behaviour is arguably more damaging to a publisher than to an author. This is particularly true where a publisher has such a strong brand image as that enjoyed by O'Reilly. Once an image becomes tainted by too many mediocre books spiced with some really bad ones the brand loses its appeal.

The other day I was thinking about the kind of books that get published for software developers. There is an embarrassing amount of junk aimed at the newcomer. There is also a small number of exceptionally good books that are well suited to the expert. However, where are the books for the improver? By that I mean the reader who can already manage the basics of programming despite all the hurdles placed in the way by the poor quality of books for novices, but needs to move on to meatier stuff. Most excellent books such as those published in Bjarne Stroustrup's 'C++ In Depth' series are beyond the ability of the average programmer. Even the works of such excellent authors as Scott Meyers are ill suited to the basic programmer. So do you have a book in you? One that the lower levels of working programmers could read profitably?

Francis

The following bookshops actively support ACCU (the first three offer 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 me know so they can be added to the list

Computer Manuals (0121 706 6000)

www.computer-manuals.co.uk

Holborn Books Ltd (020 7831 0022)

www.holbornbooks.co.uk

Blackwell's Bookshop, Oxford (01865 792792)

blackwells.extra@blackwell.co.uk

Modern Book Company (020 7402 9176)

books@mbc.sonnet.co.uk

An asterisk against the publisher of a book in the book details indicates that Computer Manuals provided the book for review (not the publisher.) N.B. an asterisk after a price indicates that may be a small VAT element to add.

The mysterious number in parentheses that occurs after the price of most books shows the dollar pound

conversion rate where known. I consider a rate of 1.4 or better as appropriate (in a context where the true rate hovers around 1.5). I consider any rate below 1.25 as being sufficiently poor to merit complaint to the publisher.

“Not really a book review”

Refactoring: Improving the Design of Existing Code by Martin Fowler and others (0-201-48567-2), Addison-Wesley, 431pp

A sort of book review by James Roberts

The August 2002 Overload magazine prompted me to write this article. It's not really a book review and I would like to see some feedback on other people's opinions on both this book and the subject of refactorisation as a whole.

I noticed that the word 'refactoring' was used both by Jon Jagger (Section heading in 'even more Java Exceptions') and in a more detailed way by Allan Kelly (Philosophy of Extensible Software). The latter referenced this book which had also been recommended to me by a work colleague, so I decided to read it.

The book takes as its subject the process of improving codes internal structure/readability etc. without making any changes to the external working of the code. From a user's point of view, the refactoring process should have no effect. From the maintainer's point of view, this process should make the code easier to understand, debug and modify.

Although the book is nominally by Fowler, there are some major contributions made by Kent Beck and others. I was surprised how consistently the style was maintained throughout the book, which is written in a chatty, easy to read style. I found it generally both clear and easy to read.

The book attempts to cover a number of purposes:

- explanation to the reader why refactorisation is a good thing to do
- a 'catalogue of refactorisations' – a set of instructions of how to perform refactorisation.
- a survey of refactorisation in the real world.

To be honest, I had problems with all of these sections.

Although I agree that there are circumstances when refactorisation might be a good idea, there were some things that I disagree on:

- A solid unit test suite will make it safe to change code. If this were the case, code would never have bugs in it. Yeah, right. This hint is given: 'It is better to write and run incomplete tests than not to run complete tests'. True, I agree. However, it would be folly to rely on these incomplete tests to protect you against stupid errors introduced by your (functionally unnecessary) code changes.
- If your manager/team leader does not believe in refactorisation, do it on your own initiative without telling what you are doing. Although this might be sometimes necessary it does seem dangerous to advocate it without major caveats. In particular, if you are going to put this refactored code through code review (as

recommended by the book), how are you going to stop your team leader noticing the refactoring. What is his reaction likely to be when a 20 line code change has turned into a 200 line code change (sure, 180 lines are trivial changes, but the difference utility will still quote them). I have had an experience where a coder reordered a source file so that all the methods were in alphabetical order. The code review is non-trivial under such circumstances!

- There seems to be a thread of 'good up-front design is not necessary, as we can always refactor the code afterwards to batter it into a good design'. This might be true at the micro-level of system design. I don't agree at higher levels.

A large part of the book consists of the refactoring catalogue. These are a set of instructions to be used to perform particular refactorisations (for example 'Replace Type Code with Subclass' which explains how a switch statement can be replaced with subclassing). This is written with a Java programmer in mind, although it is general enough to be generally applicable to C++ (for example). In general I found these recipes probably over simplistic (nothing there that I would not do instinctively, I think), while giving a false sense of dependability.

Things I didn't like (from a brief survey)

- compile and test not being the last code change in at least one recipe (Inline Method).
- In the recipe: 'Replace Type Code with Subclass' a C++ coder might forget to check that the class being subclassed has a virtual destructor. (Possibly not an issue to a Java programmer?)
- there seems to be no acknowledgement of working in teams. A change to a method name might make sense to you, but might not be as clear to a colleague. (Or might just irritate the pants off them if they happen to have remembered the names of methods in particular classes).

Towards the end of the book, a chapter is devoted to the subject of why coders do not refactor.

Interestingly, my concerns of whether unnecessary code changes can be made safely did surface. The solution here was an automated refactoring tool for Smalltalk. Apart from the fact that I wouldn't trust this any more than a coder (unexpected side-effects, timing issues, language peculiarities, subtle bugs in the tool) what does this say for Java refactorisation?

In summary, I found this an interesting book, with some good points and it was a useful exercise to read it. My main grouse was not in whether refactorisation is a 'good thing' or not (I believe that it has its place), but that I felt that it might encourage a 'gung-ho' attitude in a reader towards refactorisation, giving the impression that there are code changes that can be made purely mechanically, without care, attention, planning and thought. Looking at Amazon's book reviews, I realise that it might be a rather a contrarian point of view to not love every paragraph of this book. I wonder what other people's opinions are?



Practical C++ Programming 2ed by Steve Oualline (0-596-00419-2), O'Reilly, 548pp @ £28-50 (1.40)

reviewed by Paul F. Johnson

The second edition of any book should really be fixing the problems of the first edition. This book breaks the mould and even makes some of the problems worse!

It is difficult to say where this book is of any real use to someone starting programming. The code examples are incorrect, descriptions are incorrect (or glib or plain wrong), end of chapter exercises don't bear fit the chapters (material not covered in the chapter or just unclear) and there is a lot in there that shouldn't be.

A book on C++ programming should concentrate on the language, not extensions (`w_char` and `w_string` appear nowhere in the standard), IDEs, makefiles or the compiler. None of these have anything to do with the language, so why does this book spend any time on them? Completeness? Well if that is the case, why before the shots of the Borland IDE, is there a standard "Hello World" with the usual `int main()` and yet the screen shots have `#pragma` and change the form of `int` to `int main(int argc, char *argv[])` without any explanation? This will only cause confusion to the newcomer.

The use of the C headers are confused. While `cstdlib` is used correctly, whenever `assert` is used, the text has `#include <assert.h>?`

One of the most powerful aspects of C++ is that of namespaces – yet the subject is glossed over. Namespaces may not be easy but they are essential. The `std::` namespace can be missed if you're not careful!

Possibly the worst part of the book are the number of broken code examples. This is a beginner's book. The last thing they want is for the code to result in compiled binaries that give incorrect results. Very near to the start (shortly after the `float` variable type is introduced, there is a code example for calculating the area of a triangle. The formula given is

```
int area = (height * base) / 2;
```

Okay if height and base result in a number divisible by 2 and are integers. Failing that, the answer is incorrect. Why did the author not just define area as a `float` (or better still, a `double`)?

More of these elementary errors follow...

Examples with `main` as the first function calling other functions which have not been prototyped before hand, using global variables for no apparent reason and a glossing over of `for`.

Almost as bad as the code errors is some of the text.

Take the following

```
#include <iostream>
// This function won't work
void inc_counter(int counter)
{ ++counter; }
int main() {
    int a_count = 0;
    // Random counter
    inc_counter(a_count);
    std::cout << a_count << '\n';
    return (0);
}
```

Firstly, all of the above is valid code. The `inc_counter` function does work. It works perfectly. If a debugger is run in conjunction with the compiled code (which the book later recommends), then in the `inc_counter` function, the debugger shows `counter` has been incremented. `a_count` is not a random counter. It's a variable. For it to be a random counter, it has to be set to a random value.

What the author is meaning to say is that the code does not work as was intended. That does not excuse the fact that his use of terminology was wrong. This happens quite a fair bit as well, so it is not an exception, it's almost the rule!

The programming exercises seem to be answered at the end of the chapter as the numbering is the same. Hmmm. Hold on, these are not the answers to the same numbered questions, even though it says they are the answers to the questions.

Programming style is highlighted throughout the text with such gems as

Now consider this program fragment:

```
if(count < 10) // If #1
    if ((count % 4) == 2) // If #2
        std::cout
            << "Condition : White\n";
    else // indentation incorrect
        std::cout
            << "Condition : Tan\n";
```

There are two `if`-statements and one `else` to which `if` does the `else` belong to? Pick one

1. it belongs to `if #1`
2. it belongs to `if #2`
3. you don't have to worry about this situation if you never write code like this.

The book's answer : 3. The correct answer is 2.

While 3 is partially correct, it doesn't help a beginner as the answer isn't actually given.

All of these problems were up to Chapter 9 – and I've only highlighted them in general! I cannot bear to read past there as this book should be firmly consigned to the book shop shelf to gather dust. Any beginner who has bought this should use it to prop up an uneven bed and get hold of a good C++ beginner's book. [*Or take it back to the bookshop and demand a refund. FG*]

While it is not the worst C++ beginner's book (I still believe that honour goes to Herb Schildt who's crimes against good programming techniques and correct code can not be over expressed), it is far down the list. Even the latest edition of C++ for Dummies is better. [*Actually I think it is much better but that is personal opinion. FG*]

A definite blot on the O'Reilly book portfolio.



C++ in a Nutshell by Ray Lischner (0-596-00298-x), O'Reilly, 791pp @ £28-50 (1.40) reviewed by Francis Glassborow

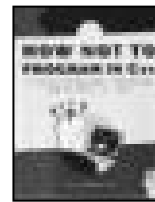
This is by far the best book that O'Reilly have produced on C++ (but then neither I nor the other ACCU reviewers think much of O'Reilly's offerings in this area). Working at my own desk I far prefer my other reference books (such as The C++ Standard Library by Nico Josuttis) however if I had to travel away from home and needed to keep the weight down but still have a good memory jogger with me, this is a book that I would seriously consider.

If you want to check the parameters of a standard function, remind yourself about the

public members of an STL container or check if something is part of Standard C++ then this book could be just what you want to take with you.

This is a reference book through and through covering all aspects of the core language and library. I have no doubt that I could find a few typos and errors if I really put my mind to it, but that would be true of just about any book ever published. However most of the problems would seem to be with either lack of detail or poor quality in the index. Let me give you a single example of this. I looked up virtual destructor in the index. I was referred to page 155 where all I found was a mention that the implicit destructor of a derived class would be virtual if the base class destructor were virtual. As far as the index was concerned, that was it. Hunting around turned up a section entitled Polymorphic Constructors and Destructors that had a little more detail but gave an indeterminate reference to something earlier. Now such back references are fine in books designed for reading but they are not adequate in a reference book where a page number should always be given.

This book, however, gets much closer to the standards on which O'Reilly have built their reputation, it is a good reference and good value for money. Perhaps they can persuade the author to work harder at cross referencing for the next edition.



How Not to Program in C++ by Steve Oualline (1-886411-95-6), No Starch Press, 265pp @ £14.06 from amazon.co.uk reviewed by Francis Glassborow

There is a profound irony in the title of this book because judging by the review of the author's second edition of Practical C++ you might think this title was appropriate to that book. Indeed readers of that book will find much in this book will illuminate some of the bad experiences they have had as a consequence of reading it.

This book is orders of magnitude better in both design and concept to the other books from the author. Perhaps this is because what is a vice elsewhere is a virtue here. The substance of the book is 114 short programs of which all but 3 are in the opinion of the author, profoundly broken even if they compile and execute.

Now I am not sure to what extent pure C programs have their place in this book, and I am not sure that a program that outputs an octal result when a decimal one was intended should actually be described as broken but let me leave that aside. What I like about the book is that it makes the reader think and has been designed to avoid accidentally seeing an answer. At the end of each program the author provides a hint number and an answer number. Neither the hints nor the answers are in the same sequence as the problems. Some hints provide references to further hints (there are a total of 361 hints). But he also provides 115 answers. As far as I can see no problem is supplied with two answers so which answer does not have a problem? Then there is the matter of the chapter with the three working programs. Well problem 103 isn't a program, problem 104 needs the ability to read minds and problem 105 would be better in an obfuscated C contest.

There are numerous other problems with the author's solutions. For example one problem assumes that MSDOS based compilers always used 16-bit ints. While most did that is not true in all cases and how would a modern programmer spot that? Of course the problem could have been rephrased so that it was stripped of OS specific references.

Now, despite my carping I like the concept of this book and the execution of the concept is not bad either though it could have done with a thorough technical review to ensure that problems were set carefully and that answers were free of subsequent errors. I think that some of the problems should simply have been thrown out (problem 5 is certainly not a broken program and would output exactly what you might have expected in some environments (some systems would add a carriage return as part of the exit procedure from a program. If you do not suffer from high blood pressure and do not mind some problems being defective this book could while away the odd hour. The sad thing is that if the author had ensured that all the broken programs were actually broken the challenge would have been greater. Instead I have a continual doubt that the problem program I am looking at is in fact broken and that spoils the challenge.



Teach Yourself Visual C++ .NET in 24 Hours by Richard Simon & Mark Schmidt (0 672 32323 0), Sams*, 414pp @ £21-99 (1.36)

reviewed by Paul S Usowicz

I find it quite humorous when I see these '24 Hour' books. Surely no one can possibly expect to learn something as complex as Visual C++.net in just 24 hours. In truth, this book should have been called 'Upgrade your existing Visual C++ 6 skills to Visual C++.net in 24 hours'. That would have been more descriptive but would not have excused the poor quality of the book.

In chapter 3 you are taken step-by-step through an MFC application and then a .net application so that you can see the differences. Sadly I could not get the .net application to compile. For a beginner this would be mortifying. I was just frustrated. At the end of the chapter I was told that I could use the code that was included on the CD. The only problem was that there was no CD shipped with the book. The back cover revealed that all of the source code was actually on the web site. After finally finding the relevant zip file I was amazed to find that the code I couldn't compile was not even in the zip file! I could easily understand if most people gave up here and just put the book in the bin.

I did, however, persevere. As I carried on with the book two things became clear. Firstly, this book is not for a beginner. It may be for beginners of the Visual C++.net compiler but not the language. A good C++ understanding is essential for this book. To be fair, this is stated within the introduction but not on the cover, or rear cover, of the book. Secondly, the book is (in my opinion) very poorly written and hard to follow. I did pick up some useful information but found it very hard work to go through the book and was quite glad when I had finished.

In conclusion I would not recommend this book to a beginner of C++.net and would only suggest it for Visual C++ 6 users if you are very short of cash and cannot afford a better offering.



Microsoft Visual C++ .NET Professional Projects by Sai Kishore & Sripriya with NIIT (1 931841 31 4), Premier Press, 1026pp @ £36-99 (1.35)

reviewed by Paul S Usowicz

I liked this book. I liked this book a lot. This book is aimed at me. I have been developing with Visual C++ 6.0 since it was available and have now started using Visual C++.net.

Unlike most books that tout 'professional' in their title this is definitely one book that lives up to one definition of the word – 'produced with competence or skill'. The topics covered include the usual stuff you would expect in a .net book: assemblies, GDI+, ADO.net, ASP.net, etc. The difference here is that they are used in what are extremely useable base applications that with a little work could form the basis for some very good commercial products.

I found the book very easy to read with uncomplicated instructions, simple explanations and very clear diagrams where needed. I found I would get engrossed in a chapter and just sail through it typing in code if I felt it was needed. Having said that, the code is available to download from a web site. Now here is my niggle (yes, just one). The code is downloadable for each project. A bit annoying as I prefer to download one file rather than 13. I then realised that this was due to several of the files being over 1MB. This would have made one file over 4MB and may have put people off downloading it. Imagine my amazement when, upon opening one of the large files, I found that most of the expanded content was taken up by an intermediate file that was not required to compile the code! It was a shame to have the whole package let down by one small annoyance.

Would I buy this book? Yes. It would be hard not to with a very reasonable £36.99 price tag. I have paid much more for much less in the past. If you program with VC++.net for a living then you should really investigate purchasing this book.



Inside Microsoft Visual Studio .NET (2003) by Brian Johnson et al. (0-7356-1874-7), Microsoft Press*, 542pp @ £36-99 (1.35)

reviewed by Ralph McArdell

I was looking forward to reading Inside Microsoft Visual Studio .NET (2003) as I have a need to extend Microsoft Visual Studio .NET – the focus of the book.

In the main I was not disappointed and even picked up some everyday usage hints. Most subjects are covered in detail, occasionally to the point of being slightly tedious. These mostly concern macros and the various forms of add-ins and topics related to them and include customising Visual Studio .NET help and designing setup projects, useful if you plan to distribute your add-ins.

In one or two places the book is less detailed, describing a solution based on the downloadable code that accompanies the book. This was understandable as these cases tended to require knowledge beyond the scope of the book such as writing ActiveX controls.

I found the material generally easy to read and understand but the text was marred by several obvious typos. Although covering the latest 2003 version of Visual Studio .NET most of the material also applies to the 2002 version.

I particularly liked the notes and asides on tips, bugs and pitfalls, which will no doubt save hours of frustration.

On balance I think this book will prove to be as useful as I had hoped and I would recommend that anyone who wants or needs to know more about extending and customising Microsoft Visual Studio .NET take a look at it.

Java & C#



J2EE FrontEnd Technologies by Lennart Jörelid (1 893115 96 8), Apress, 1112pp @ £43-00(1.39)

reviewed by Stephen Dicks

The front cover of this book had my alarm bells ringing at the start – Apress appear to have trademarks on the phrases 'The Experts Voice' and 'what you need to know'. Highly involved with `javax.com` he may be, but the author of this 1,000 page book left me with just one question – what was the point?

This book is about the technology associated with J2EE – servlets, JSPs, EJBs et al. Having got that as a target, I suppose the book actually describes the technology quite well; although the execution of a unix command line at one point (to execute ping) when at least some of the book feels quite NT-orientated might throw off the beginner inclined to 'type in and test' the code on that platform.

My real complaint about this book is its lack of context; it describes the technology in a 'simplest program possible' approach, without really getting underneath the whole purpose to the activity. This is brought home most clearly via the chapter on Apache Struts which is only 80 pages, when 250 pages have been devoted to plain old JSPs; the author acknowledges that Struts 'starts paying off in well-constructed web applications' but can't be bothered to describe it in any detail, preferring to stay on the safe ground where he has obviously trod the boards for several years.

Also although the book claims to be up-to-date on the various Servlet APIs, he fails to mention the subject that I personally struggle to find material on, that of creating WARs (web application archives) although the appendix devotes 13 pages of text and screenshots showing how to install one using the reference EJB implementation.

I also felt throughout the book that not enough emphasis is put on the distinction between elements that are part of the servlet specification and those that are container specific – this book is not aimed at programming neophytes and so (given the name dropping of several J2EE vendor implementations) I found it unreasonable that the author couldn't find space for re-implementing in each container where appropriate.

Maybe I expected too much from this book; but then again the author is described as an 'expert' and the front cover promises 'develop the knowledge needed to create successful enterprise applications'.

Ultimately disappointing, I wanted much more detail on the leading-edge technology (like struts) and fewer forests destroyed for the sake of tired simple servlets and JSPs.



EJB Design Patterns by Floyd Marinescu (0 471 20831 0), Wiley, 259pp @ £25-95 (1.35)

reviewed by Stephen Dicks

The title of this book had ‘buzzword bingo’ written all over it and the foreword (by the CEO of the company employing the author) does nothing to quell that fear by describing the author as ‘the World’s leading expert in EJB design patterns’.

Despite this lavish introduction, the first half of this book actually delivers pretty well what it says on the tin; a Gang-of-Four (GOF) specifically for EJB developers. A number of the GOF patterns are revisited here (e.g. Command, Facade) but they are placed into the EJB context well. The level of example code to text and diagrams is very well thought out, with just enough code to allow simple application of the techniques discussed.

However somewhere around half-time the author embarks on a second completely different book, covering the ant and JUnit tools and discussing alternatives to entity EJBs. This is then compounded with a few minor niggles in the code examples – the EJBHomeFactory implementation has the author in angst over thread-safe initialisation of the singleton. (Unfortunately he appears to have not come across the double-locking optimisation).

I would have to recommend the first half of this book to any EJB developer; unfortunately the second half (and its copious code listings) let the side down somewhat. If the book’s title had allowed some servlet coverage then it might have allowed a sensible page count with the same quality as the first half. Roll on a second edition with maybe some servlet patterns and the 2nd half deleted.

Recommended (just).

Highly recommended for all but beginners.



NetBeans: The Definitive Guide by Tim Boudreau et al. (0-596-00280-7), O’Reilly, 646pp @ £31-95 (1.41)

reviewed by James Gordon

NetBeans is a free to use Java IDE from SUN and for the boys to get a book printed by O’Reilly is a real bonus. Is the book warranted? Well I think the answer is yes.

It starts with downloading and installing the JDK and IDE and even building NetBeans from the source. The book takes each different area of Java development and explains how to do it in NetBeans. These include Beans, CVS, compilation, Javadoc, XML and JSP modules.

Well that is half the book, the other half is to do with extending the IDE with your own modules detailing the APIs and the ‘New Module Wizard’ and the internals of running a module in NetBeans.

This book is really an end-to-end tutorial and reference book for using and expanding NetBeans. If like me you only use NetBeans the book is still a bargain. I struggled for a while with creating a Bean, two nights of reading the Beans tutorial and I’d cracked it.

There is so much that NetBeans does for you, wizards that make jobs easier and maintenance easy. I’ve missed most of them and have only found them and NetBeans real power by reading sections of this book.

If you’ve got NetBeans then get this book.



A C# Application from Inspiration to Implementation by Kyle Dunn (1-86100-754-X), WROX*, 360pp @ £36-99 (1.35)

reviewed by Paul F. Johnson

If you are planning on writing a small scale application in C#, then you won’t go far wrong with this. It covers everything from the initial meeting with the client to the packaging of the final application using a deployment tool.

The design process and ideas behind it were very clear. It has been a long time since I last came across a book that paid such attention to the processes that precede writing the first line of code.

Before a line of code is written, it’s chapter 6.

That’s when the book falls over – the code.

It’s not so much “here’s the application code that I used” more “here’s some of the application code I used”. Unfortunately that problem really does detract from the book as it means that you cannot compile the application so the results can be seen.

A second problem with the “some of the code” approach is that if it had been “all of the code” then the book would have served a dual purpose as it could have also been one of the finest Visual .NET how to books around; the book pays closer attention to the setting up of the forms, setting out of the database (which is an MSDE one, though it is outlined as an SQL one) and other aspects of using the Visual development environment.

The book would have benefited from having a CD of the source code and other components – if for nothing else than for the reader to be able to see the development as it went along and not have to download the .NET deployment system.

A great opportunity for being one of the best sadly missed.

Visual C# .NET A Guide for VB6 Developers by Brad Maiani et al. (1 861007 17 5), WROX*, 530pp @ £28-99 (1.38)

reviewed by Sue Heathcote

This is a typical Wrox press book that has been written by a collaboration of authors (in this case 5). The writing style does not appear to have suffered as a result and is consistent throughout the book, being a style that is easy to read. This is a book written specifically for Visual Basic programmers who wish to migrate to C# quickly.

Covers .NET v1.0 and you will need Visual Studio.NET professional or higher with either SQL server or an MSDE database server in order to work through the examples. (Visual C#.NET standard edition can be used but not all the projects in the book can be run using this edition). All the code is available to download from the Wrox press web site, along with any updates to the book. (This is not a facility that I have tried).

The book assumes the user has a good working knowledge of Visual Basic and Windows programming in general, so it does not attempt to teach the basics. This means that example programs are being created from the opening chapters of the book with little explanation of how things work.

Chapter 1 is an overview of the .NET environment and how Visual Basic.Net fits in. This is rather a heavy topic if you have no prior knowledge of the .NET environment. Chapter 2 jumps straight in with a working Windows

program of the classic Mastermind board game, which provides a little more meat than the average ‘hello world’ program. Many topics are introduced briefly with the explanation that ‘we will explain this later’, which can be a little frustrating if you are itching to get under the hood.

Chapter 2 delves into the C# language itself. Describing all the basic constructs and detailing the differences between C# and Visual Basic. This is well done and gives a good understanding of the differences. Some of the language constructs are very similar to Visual Basic and should be well known to any Visual Basic programmer and in this case the descriptions can be a little too detailed, but it is easy to skip over these to the next sections.

The middle sections of the book are devoted to Object Oriented programming. This covers classes, constructors, inheritance, overloading, abstract classes, interfaces, static members, delegates, events and much more. If you are new to OO programming then there is sufficient detail to get a good overview of the subject and its use in C#. If you have used OO (say in C++ programming) then there is nothing new in this section, except how the concepts are dealt with in C#.

The later chapters cover the .Net framework, Active X controls, data access with ADO, integrating VB6 and C# and deployment of applications. The .NET framework is mentioned about half way through the book, but not in very much detail. If you are looking for a book showing how to use the .NET framework then this is not a book for you.

This book is aimed directly at the Visual Basic programmer who has been using VB to develop user interfaces and simple programs and has never really delved into the concepts of classes and objects. For this type of programmer it gives a good introduction to the C# language and its differences to VB and explains how these new concepts work. If you are an experienced OO programmer, or a VB programmer who has extensively used the class and interface features of VB, then I do not believe that this book is for you. It would provide a good introduction to C# programming, but would leave you wanting more information.

C# Unleashed by Joseph Mayo (0 672 32122 X), Sams, 794pp @ £36-50 (1.37)

reviewed by Paul F. Johnson

Sams in the past have come into a lot of criticism for their C++ in <insert unrealistic number> Days and most of the time, it is well founded. As such, they have had a rough ride in the book reviews.

This time, they have a fantastic book. It makes one assumption (which is fair enough given that it is aimed at intermediate/advanced level) and that is that you’re coming from a C++ or Java background. That said, if you’re fresh in with a bit of brains, you should be able to follow the book.

The book covers the language in great depth with plenty of good examples with clear, concise explanations of the code. Every aspect of the pure language is covered with tips and hints for C++ and Java programmers.

It is not a light book, weighing in at a mighty 820 pages (or so) and many would be thinking that it’s quantity over quality. This is not the case

here. Take the sections on file handling. An entire chapter is given over to this subject (around 40 pages) with even more later on in the book. Even in some of the best C, C++ and Java books around, I have not seen this much given over to the file handling routines.

As it is the pure language that is covered (for the most part), the book is an invaluable resource for those using GNU .NET and Mono. Towards the very end of the book, it begins to cover the Windows specific parts, however, these parts are described and explained as well as the pure language part, so when the Windows parts are finally available, the use of these parts will be have already been covered sufficiently.

The source code from the website is clear and well documented.

There is only one drawback with the book – it doesn't tell you how to compile the source code from the command line; it is assumed that you will be using the Visual Studio IDE (and debugger). While I don't have that much of a problem with this as such, having the command line compilation command would given the book that little bit extra.

Other Programming

The Art of Unix Programming by Eric Steven Raymond (0-13-142901-9), Addison-Wesley (Free on-line copy: catb.org/~esr/writings/taoup/html/) reviewed by Vaclav Barta

Eric Raymond, a well-known Open Source advocate, set out to 'capture the engineering wisdom and philosophy of the Unix community'. He was remarkably successful.

Books describing the architecture of a piece of software are not very common and the author clearly intended his work to be one of a kind. As an Open Source enthusiast, he also strived to use Open Source principles in the book's creation wherever possible. He solicited cooperation of a number of leading Unix developers and kept the evolving manuscript on-line, inviting comments from the general public. (This is a review of the on-line version, version number 0.66 – I've never seen the printed book, which should be published in August 2003. I also did not comment on it until now.)

The amount of work that went into the book shows off in extensive and authoritative references, numerous case studies (real programs only, with an emphasis on but not nearly confined to author's own Open Source software) and also in the elaborate book structure. One of the more unusual stylistic devices it employs is the in-line preservation of some comments about early versions of the text, with attribution, which proves an effective way to highlight design controversies. Some choices in the book structure (e.g. putting off a section about Unix documentation until chapter 18) feel arbitrary, but overall, it is presenting a logical progression for readers who want to read even technical books from beginning to end while giving clear instructions on what to skip according to individual preferences and prior knowledge.

Part I, Context is a general introduction, Unix history and comparison to other operating systems. It mostly repeats what the author has written on previous occasions. Extensive Microsoft bashing ('Microsoft's mal-engineered

software was rising around us like a tide of sewage') serves to remind the reader that this is an advocacy book: the author has an agenda and wants to convince you. I hasten to add that this is not a criticism of the book – there's nothing wrong with having opinions and Raymond's opinions and advocacy are well known – but merely a warning not to read the book uncritically. I did not find any factual errors. The tone of the writing did compel me to look.

Part II, Design discusses the usual keywords (modularity, transparency, complexity...) as well as topics I've never seen in texts about software design, for example Unix conventions for configuration files. These 'unusual' chapters I found most interesting. The case study on Emacs vs. other editors (where Doug McIlroy, one of the guest contributors, sharply disagrees with Raymond) recapitulates a well-known polemic still going strong after decades of successful deployment, re-implementation and refinement. The accompanying case studies show genuinely new architectures for a Unix editor – many Unix designers clearly do not consider the fact that something works a good enough reason to stop improving it...

Part III, Implementation describes Unix programming languages and tools. The technical parts are especially suitable for a beginning programmer, or one who didn't program under Unix before. The chapter on re-use and Open Source licences is a good example of the author's writing methods and advocacy goals. He feels strongly about the importance of Open Source software licences and – perhaps guided by early reactions to the section (the old version I remember reading was much drier) – spares no rhetorical effort to get his point across. I don't think I'm entirely convinced, but it was worth reading.

Part IV, Community discusses documentation – including an introduction to XML and DocBook – software standards and Open Source best practices and then gives an overview of the current technical issues exercising the Linux community.

Conclusion: For the question 'Why are we using Linux?' – whether it's posed in an educational or commercial software development context – this book gives some very good answers. If you are (or want to become) a professional programmer and want to acquire thorough understanding of Unix as a software development platform, then – independently of your current knowledge of Unix in general and Linux in particular – you should at least skim this book. Highly recommended.



XML Schema by Eric van der Vlist (0 596 00252 1), O'Reilly, 380pp @ £28-50 (1.40) reviewed by Rob Hughes

There is something about all things XML that I find makes the subject a tricky one to break into. The difficulty is, I suspect, more due to the complexity of terminology than the general complexity of the subject area. Despite this, I found this book approachable and more importantly, useful.

The text progresses logically from introducing the reader to the basic concepts to the deeper and darker areas of XML Schema. The author is not afraid of showing different approaches where appropriate and discussing the benefits and

drawbacks of each. There are also frequent interesting insights into the background of Schema, the ways in which it differs from the less expressive DTDs and the problems inherent with the current Schema language. The final quarter of the book is basically a reference, so it is valuable both to learn from and as a manual.

The style of this book is exactly that which I like to see and enjoy reading. The writing is brisk and concise yet maintains clarity, which helps to keep the book approachable. The style is well supported by a structure based around short chapters; each of which is digestible in a reasonably short time period.

Although I personally find the subject area awkward and non-intuitive, if you need to work with XML Schema, I recommend this book as a very useful guide and reference.



XSLT Cookbook by Sal Mangano (0-596-00372-2), O'Reilly, 654pp @ £28-50 (1.40) reviewed by Rick Stones

This is another book in O'Reilly's 'Cookbook' series, which borrow from the ideas of patterns and present a series of problems, solutions and explanations of the solutions. The problems are grouped into chapters such as 'Dates and Times', 'Selecting and Traversing' and so on. There are just over a hundred recipes in this book, ranging from some just a few lines long, to a few that are three pages long.

I found the level of this book a bit higher than the Perl Cookbook (an excellent book) and the examples longer. If you are a beginner to XSLT you will find all but the simplest examples somewhat challenging; at the other end of the scale I think even experts will find some of the transforms presented challenging. The preface says that the final chapter 'pushes the XSLT envelope' and I certainly would not disagree.

Apart from that caveat to beginners, this book is a powerful reference work. The problems tackled are well chosen, the solutions elegant and the explanations carefully and clearly explained, though as before not aimed at novices. The author also points out where alternative solutions exist and explains what makes a 'better' solution. Some of the solutions appear in two or more forms, for example there is a simple XML to CSV converter specific to the source file, which is very easy to follow, but this is followed by a generic, general purpose transform, which is in turn followed by one handling sparse mapping. This layered approach to presenting ever-improving solutions helps considerably with the understanding.

Methodologies & Practices



Object Design: Roles, Responsibilities and Collaboration by Rebecca Wirfsck and Alan McKean (0-201-37943-0), Addison-Wesley, 390pp @ £37-99 (1.32) reviewed by Silvia de Beer

This book focuses on the practice of designing objects as integral members of a community where each object has specific roles and responsibilities. The book does not require any experience as a developer and can be used as a first study of object design. However, also for

experienced developers this book does offer very valuable and refreshing advice and teaches about modern views on object design.

The purpose of a design process is to create a collaboration model, which can be developed by studying the collaborations and responsibilities of objects. The design process starts with writing a brief design story, searching for candidate objects and choosing good names. A whole chapter is dedicated how to find objects and valuable advice is given, which goes beyond the advice of identifying noun phrases. A useful concept while creating the collaboration model is to look at an object's role in the light of stereotypes. These stereotypes help to discover the collaborations between various types of objects. An object can be an information holder, structurer, service-provider, controller, coordinator, interfacier. One should write the object's purpose and stereotype on a CRC card and start to connect the object candidates.

I was mostly impressed with the chapter on Control Style, where in an example is shown how an implementation based on a state pattern can be converted to a delegated control style. I found this example highly valuable, because in my experience less experienced developers quickly tend for a centralised control style. With this case study many important points about different control styles become clear.

I liked the layout of the book, which uses blue section titles and useful side comments on the main text in the borders. The only negative point is may be that the chapters have no numbered subsections, which make it slightly difficult to see a hierarchic structure in a chapter.



Secure Coding Principles and Practices by Mark G. Graff & Kenneth R. van Wyk (0-596-00242-4), O'Reilly, 200pp @ £20-95 (1.43)

reviewed by Francis Glassborow

Clearly the subject of this book is going to become increasingly important as time goes by. I am sitting here working on one computer with a second one beside me so that I can try things out and a third one hiding under my desk acting as the household's gateway to the Internet.

Downstairs my wife is using her machine to look some information up. She does not have to ask me to switch on the Internet gateway; it sits their 24/7 supplying whichever of us wants access. If I want some details about a book I just use my web-browser to go out and find it. But while we are free to go out onto the Internet others have ways of getting in. Of course I have firewalls, intruder detection and virus protection software in place but like the locks on by doors and the burglar alarm that is switched on when we are out all the protection just makes things harder for the miscreant it does not provide 100% protection.

The first line of defence is that the software that runs my machines and lets me do the things I want should not be easily perverted. Ten years ago most of us just had to ensure that the software and data we loaded into our machines was OK. This is no longer the case. Quite apart from the damage that can be done to my own data there is the way that my equipment can be subverted as a tool to do damage elsewhere.

What I am saying is that security has become a major issue for all of us. A programmer who does not take the issues seriously is at best

incompetent and at worst criminally stupid. We have to start taking responsibility for our work. It is not enough to try to write bug free software, if we write software that is going to run on a networked machine we have to do so in ways that make it hard to exploit. Just because neither the software nor the hardware will be used as a direct part of a high integrity system is not an excuse.

The very least you can do is to spend some time studying the principles and practices of producing secure code. This book is one of the ways you can do this. It is not a complete solution but it is a start. The authors are well aware the problem is far more than just a technical one. Of course companies are reluctant to spend the resources to improve their software security but that is at least in part because they do not understand the issues. Then there is the mindset of programmers who simply do not believe that their work could so easily be subverted or that anyone would be interested in doing so.

While this book is primarily aimed at the software developer, it is short enough that it should also be read by managers and clients. The managers so that they are willing to spend what is necessary to address issues of software security and the clients so that they start including realistic security requirements in their specifications. Can you imagine a builder leaving locks off the doors in a new house? Well why should our machines sit out there to be invaded by anyone with the wish to do so.

If you are involved in software development either as a producer or a consumer you need to take issues of security seriously. If you have a reasonable level of technical knowledge you should read books such as this one.

Databases



Oracle in a Nutshell by Rick Greenwald and David C. Kreines (1 0-596-00336-6), O'Reilly, 906pp @ £35-50 (1.41)

reviewed by James Roberts

The stated aim of this book is to provide an Oracle reference manual that 'can be lifted without mechanical assistance'. This book just about manages this on both counts.

Although it covers Oracle 9i, it carefully labels features that are new for this release and Oracle8i, making the book useful for users of older versions. This book covers a great deal of ground and is generally arranged logically with a clear and concise writing style. In particular I found the treatment of the configuration section useful as it grouped the configuration items into related sections rather than listing them alphabetically. I also found the Data Dictionary much improved over the on-line documentation in this regard.

Although this book weighs in at a little over 900 pages, it suffers from being over-condensed. I particularly noticed this in the SQL*Plus section, which is a truncated version of the SQL*Plus Pocket Reference book (literally – the phrases are word for word the same). The truncation can make the book harder to read. For example, commands are condensed into a minimal number of rows rather than spread out in a more readable, but less space-efficient way. Also examples, which although strictly unnecessary, can make a reference much easier to

read, are jettisoned. The same abrupt style pervades the book. In short, this is a well-written book that provides a wealth of information, in a highly condensed form.

However, a reader who did not use many of the packages described within this reference might chose to own several smaller but less compacted books, or use the reliable but less well written Oracle documentation.

Pocket References

Oracle SQL*Plus Pocket Reference(no charge if with **another** by Jonathan Gennick (0-596-00441-9), O'Reilly, 113pp @ £8-95 (1.44) reviewed by James Roberts

[See web]

C++ Pocket Reference by Kyle Loudon (0-596-00496-6), O'Reilly, 130pp @ £8.95 (1.44) reviewed by Francis Glassborow

[See web]

UML Pocket Reference by Dan Pilon (0-596-00497-4), O'Reilly, 81pp @ £8.95 (1.44) reviewed by Francis Glassborow

[See web]

GOOGLE Pocket Guide by Tara Calishain, Rael Dornfest & D J Adams (0-596-00550-4), O'Reilly, 128pp @ £6.95 (1.43) reviewed by Francis Glassborow

[See web]

JavaScript Pocket Reference 2ed by David Flanagan (0-596-00411-7), O'Reilly, 127pp @ £10-50 (1.37) reviewed by Emma Willis

[See web]

Internet

IPv6 Essentials by Silvia Hagen (0 596 00125 8), O'Reilly, 338pp @ £28-50 (1.40) reviewed by Mark Ayzenshteyn

[See web]

Quality Web Systems by Elfriede Dustin et al (0 201 71936 3), Addison-Wesley, 318pp @ £23-35 amazon.co.uk price

reviewed by Christopher Hill

[See web]

Non-Programming

Designing Embedded Hardware by John Catsoulis (0-596-00362-5), O'Reilly, 298pp @ £28-50(1.40) reviewed by Silvia de Beer

[See web]

Building Wireless Community Networks by Rob Flickenger (0-596-00502-4), O'Reilly, 168pp @ £20.95 (1.43)

reviewed by Francis Glassborow

[See web]

Monster Gaming by Ben Sawyer (1-932111-79-4), Paraglyph Press*, 330pp @ £18-99 (1.32) reviewed by Francis Glassborow

[See web]

How to Become a Successful IT Consultant by Dan Remenyi (0 7506 4861 9), Butterworth-Heinemann, 166pp @ £29-99(1.59) reviewed by David Nash

[See web]