# Contents

## Reports & Opinions



## Dialogue



## Features



## Reviews



## Copy Dates

**C Vu 14.6: November 7th**

**C Vu 15.1: January 7th**

## Contact Information:


**Editorial:** James Dennett
76 Lawn Road,
Bristol, BS16 5BB
0117 9653875
editor@accu.org

**Advertising:** Pete Goodliffe
Chris Lowe
ads@accu.org

**Treasurer:** Bryan Scattergood
19 Bayford Place
Cambridge, CB4 2UF
01223 475468 (home)
01223 692445 (work)
treasurer@accu.org

**ACCU Chair:** Alan Griffiths
alan@octopull.demon.co.uk
chair@accu.org

**Secretary:** Alan Bellingham
020 8998 6964
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 journals of each type 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

In a break from routine, I would like to start by saying thank you to all of those who continue to contribute. My greatest fear when I took the helm of C Vu was that I'd be begging for material days before a deadline. To date that has not happened, and indeed you, the readership, have been spared the horror of having me publish my own articles. (That day will come, and then you will know of what I speak!)

### What should C Vu cover?

The first appearance of an article from ACCU's recently-formed Python SIG raised some concerns about the direction C Vu is taking, and which direction it ought to be taking. Some things are uncontroversial, I hope: C Vu is to be consistent with the goals of the ACCU, in particular advocacy of professionalism in programming at all levels, both when programming as a hobby and when employed to do so. So far, so good. The field of programming, however, is far too large to be covered by a single journal, or most likely even a single organisation. We must limit our attentions, or risk failing to do justice to any area we explore. The C and C++ programming languages remain the essence of ACCU, but professionalism dictates that we should not be so narrow minded as to neglect other tools. The question then becomes: how much effort should ACCU invest in things outside of the world of C and C++ and how much space should C Vu devote to them? If we cover Java, should we also cover C#? If we welcome a Python SIG, what about Perl? Tcl?

Sven Rosvall's letter is one opinion. In his column Francis Glassborow presents another. Surely there are more, and equally surely a solution will be found that is acceptable to the vast majority of ACCU's pragmatic membership. Please do consider what you want from ACCU, and make your voice heard. You can write on the mailing lists (such as accu-general), write to cvu@accu.org, or find you own forum. ACCU being the way it is, the decision will be made by the members who make the effort to influence it. Those who contribute material also have a large say – as editor, I only select from the material I receive and ask for volunteers to write about subjects that I know to be of interest to the readership.

### Software Configuration Management

A vital tool in every serious programmer's toolbox is (or should be) a version control system. Any company developing software without one should address that *now*. (If that's you, stop reading. Install version control before reading on.) Version control is the most basic level of the whole subject known as software configuration management (SCM), and companies such as Rational Software and Serena Software will be happy to buy their consultants expensive cars with the fees they will charge to make sure your SCM solution is working as it should. One remarkable thing, though, is how hard it is to find a version control tool that can support even a basic set of functionality with no fundamental flaws. Surely software designed to help development ought to allow effortless renaming of files – and yet most products fail to achieve even that. One major version control tool comes armed with tools to repair its database when it inevitably becomes corrupted. Even spending £1000 per developer does not guarantee that the basics work as they should; as is so common, the extra money seems to buy extra features, many of which you won't use, rather than buying better quality.

In a recent attempt to determine which SCM solution was appropriate for my company, one thing surprised me. It is remarkably hard to find good impartial information on the pros and cons of different version control systems when you get outside of the low-budget options such as Visual Source Safe and CVS. Various of my colleagues had experience of assorted other systems, but most of us happily just fall into place and use whichever system a company imposes without taking the time to learn about it in depth. SCM isn't what drives us, it's just a necessary thing (and sometimes feels like a necessary *evil*.)

Once again the punchline is too obvious: that ACCU members, between them, have a vast wealth of experience of version control and higher-end SCM systems, and with effort could collate that experience into a rich resource. Is there a willingness to do this? If so, how would it best be done? I would be happy to publish summaries of individual tools, or comparisons, or descriptions "from the trenches" of members' experiences in trying to get these tools to make development better, not harder.

### Teaching Standard C++

Experience in a number of places suggests to me that much teaching of C++ in academic institutions (where commercial interests have not driven faculties to teach Java or C#) is still covering pre-standard C++. Given that the C++ Standard has not changed since it was officially published in 1998, there now seems to have been ample time for most people in a position to teach the language to have updated their knowledge, and tools which are able to compile at least the vast majority of modern C++ are widely available without paying an arm and a leg. Surely many college teachers are making the necessary effort, and still others while falling short of this ideal are still offering a valuable service to students who would otherwise have nobody from whom to learn C++. Nonetheless, there seems to be a gulf between academia and practice, and unusually in this case it is often the practitioners who are more formally correct. This is a very real problem, because today's students are tomorrow's professionals and bad habits die hard. It is understandable that most teachers are not experts – achieving and maintaining expertise in contemporary computing is a time-consuming commitment – but there should be steps we can take to help. Maybe our student members can tell us how C++ is taught, maybe we can provide online material aimed at quickly bringing a motivated but rusty teacher up to speed on standard C++. Your suggestions are welcome.

And now, after this longer than usual editorial, on to the reports…

*James Dennett*

## From the Chair

**Alan Griffiths** <chair@accu.org>

In my last report I mentioned that our Advertising Officer Pete Goodliffe is standing down, in this report I'm pleased to tell you that Chris Lowe has offered to contribute. Details of how this will happen are being sorted out and there will be more information in the next C Vu. Thanks Chris for volunteering and thanks Pete for the work over the last few years.

By now preliminary information relating to the conference should be available. I was very pleased to meet many of you at the last conference and am looking forward to doing the same next year.

## Membership

**David Hodge** <membership@accu.org>

At the time of writing (9th Sept) we have had 150 more renewals than at the same time last year, if you are one of the early renewals then thank you. If you are reading this, and you have not yet renewed then this is the last journal that you will receive, unless you take some action in the next few weeks. If you are not sure if you have renewed, then the mailing label always contains your current membership expiry date. The current paid up membership is 776, which includes 60 new members who have joined since July 1st. We have 211 overseas members in 38 countries. For the benefit of the new members and anyone else who has forgotten, please keep membership@accu.org updated with changes in your postal and email addresses.

## Conference Organiser

**Francis Glassborow**
<francis.glassborow@ntlworld.com>

Plans for the ACCU Spring Conference 2003 (April 2nd-5th at the Holiday Inn, Oxford) are coming along nicely. I think that those attending will have the best ACCU conference so far. The line up of speakers will be second to none in the World and between them they will be covering a truly immense range of topics. I have no doubt that there will be some who will be irritated by some of the topics covered, but I think far more will be bemoaning the fact that they will often have to make tough choices.

There is still time for you to propose a talk you want to give, and this year we have built in time for less experienced speakers to give shorter (30 minute presentations). However please do not prevaricate, if you want to be a speaker you must let me know very quickly because by the time you read this I will be drafting the programme and confirming speaking slots with those already on my list. On the other hand please do not be put off by the formidable list of experienced speakers that you will see on our web site. We all have to start somewhere. At least you will know that you will not be faced with an audience of a couple of hundred experts but by a select group of people who

have chosen to come to your presentation because they positively want to hear it. Believe me, that helps when you are less experienced.

Now let me take up a few moments of your time on the subject of costs. There are always people who take one look at the costs of a conference, shudder and move on to something else. Now, if you are a student, a pensioner or unemployed I can understand (but remember that the ACCU will always do what it can to help such members, but if you do not ask we cannot help). However, if you are a professional in software development I think you are making a mistake. More to the point, I think your employer (which comes to the same thing if you are self employed) is making a mistake. There are a multitude of reasons why conferences in general and ACCU ones in particular are worth consideration. Let me focus on ACCU ones. For a hobbyist the cost of attendance is apparently high, but for the developer they are, frankly, very low. You get to meet expert practitioners who will enrich your professional skills. You will make contacts that may prove invaluable when you are stuck with a problem. Yes, it should be fun, but the technical content will also be very high. Do you ever wonder why so many world-class experts are willing to come and speak at our events? It certainly isn't for the money, because they do not get paid. Most of them do so, at least in part, because of the opportunities they get to listen to each other. I think we should not value them any less than they value each other. If you are truly a professional in software development you should be attending at least one conference a year. If you employer does not agree then it is time to ask yourself whether you really want to continue with that employer.

Hobbyists also get excellent value but for slightly different reasons. How much does your hobby cost you every year? I suspect that it is much more than you think. Many years ago when I was an officer of the Oxford University Judo Club I became increasingly frustrated by the reaction of so many when they learnt of the cost of a judo outfit. Yes it was substantially more than you would pay for almost any other item of sports clothing. So I did a survey of the cost of sports clothing for all the sports I could. The year on year costs for Judo were amongst the very lowest. Judo outfits are robust, you do not have to keep replacing them. That is more than can be said for running shoes, track suits etc. The up front one time payment was high but the long-term costs were definitely low. I think that compared to the costs of being a football supporter, climber, model train enthusiast etc. the cost of coming to the ACCU conference each year will seem pretty small. After all, be honest, how much did you spend on your computing gear last year?

I sometimes hear people say that their husbands, girlfriends etc. do not like being left at home for four days. So don't: bring them with you. There are lots of things to do round Oxford and it is close enough to London for them to spend time there as well. I can understand that it is not always possible to attend a particular conference, but if you really are not interested in attending them at all, I am left wondering why you want to be a member of ACCU.

Start your campaign to be at the ACCU Spring Conference 2003 today. Get your employer and your family on side in time to make a booking. And do not forget to encourage your colleagues to come as well.

# Standards Report

**Lois Goldthwaite**

The international C++ standards committee has a new convenor. He is Herb Sutter, well-known to ACCU members from his lectures at the annual ACCU conferences. Sutter is also the author of the books "Exceptional C++" and "More Exceptional C++", many magazine articles, and the Guru of the Week web column on the internet newsgroup `comp.lang.c++.moderated` (catch up on previous Guru puzzles at `www.gotw.ca`). In addition to his independent writing and consulting, he is also C++ community liaison for Microsoft. The convenorship involves a serious commitment of time and travel, and IMHO Microsoft deserves a lot of credit for underwriting C++ standardardisation efforts to this extent. Along with ACCU, Microsoft is one of the sponsors for the C++ committee's week-long meeting in Oxford next April. The ACCU conference on April 2-5 falls between the meetings of the C and C++ committees, and will feature speakers from both groups, so you can expect a very high level of technical content. Book early to ensure a place.

Sutter succeeds Tom Plum, who has been WG21 convenor for the past six years. (ISO/IEC JTC1 SC22 WG21 is the official name for the C++ standards committee. Try saying it real fast three times.) At the meetings and on the committee reflectors, debate can sometimes get heated, but Plum's calm leadership has kept the effort on a steady course. He will continue to participate in C++ standardisation.

# Mentored Developers

## Python Project

**Jim Hughes and Tim Penhey**

The python project is finally underway, and at the time of writing (5-Sep-2002) the fast track group should have finished chapter 2 of the text. The students of the python project decided to split into two groups. One group aiming to go through a chapter every two weeks, and the other group aiming for a chapter every four weeks. People were then free to choose their speed based on past experience and available time to put forward to the project.

The text that both the fast and slow track groups are using is: "Learning Python", Mark Lutz & David Ascher, O'Reilly, ISBN 1-56592-464-9

Chapter 2 covers the different data types. As well as the "normal" basics like strings and numbers, python also has tuples, lists, dictionaries, and files as standard types. There hasn't been much discussion on the list yet, and we have some fantastic people mentoring the groups. Although by the time you read this there should have been much more in the way of messages.

## XML Project

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

There have been some changes on the XML project front in the past month or so. A change in coordinator for the project coincided with a desire to set up a learning XML strand as well as the XML editor/validator strand. As there seems more student interest in the learning XML project, the sensible approach seems to be to concentrate on this, with the hope of gradually moving into the more advanced project as interested parties develop XML skills. The main step that the learning XML strand has taken is in adopting 'Learning XML' as a study text. Orders have been placed, and once copies are in the hands of students and mentors we will push on with developing the project further.

## Book Group

**Paul F. Johnson**

"*C++ from the top – a guide for those new to programming*"

The idea of the book came about very shortly after the mentored developers group started to get itself moving. Basically, it was to follow the "course" and at the end, become a rather good book. That was the plan.

However....

Rather than writing a book based solely on the ACCU groups activities, it was decided that a book for new comers to the language (and to those moving language over to C++). The premise was that currently most books for those new to the language have some severe problems:

1. They are revised from older versions which normally pay lip service to the C++ standard, but usually contain the same technical mistakes they previously had.
2. They are C books which have been flavoured with C++ or worse, explain C++ as if it were an extension to C rather than what it really is
3. They are usually poorly written or full of inaccuracies (such as finding void main() or iostream.h ) – giving someone new to the language such a bad start is unforgivable, especially as the standard is now 4 years old.
4. Quite a lot, if not all assume that the person has some prior language experience. Even the likes of Schildt and the "for dummies" series assume prior experience (and we all know how bad they are!). There is a definite gap for those who know how to basically switch on the machine, insert a CD and that's it.
5. Books seem to favour one platform or one compiler. This is usually for a Windows or MS-DOS based machine. This is one of the largest detractors to learning, especially if you are not using that specific platform.

From a very early stage in the development and planning of the book, the book team (which consists of Paul Grenyer, Kevlin Henney, Terje, David Nash and myself) decided that we would come up with a contents list which was not only logical, but would not scare people off. It was also decided that a CD would be available with the book with a copy of GCC for Windows, Linux, DOS, MacOS and RISC OS (this may expand to include an Amiga and Atari version, but it seems unlikely). With the exception of RISC OS, the version of GCC shipped is (currently) 3.2 with RISC OS using 2.95.4. A full set of source code will also be on the CD with explicit instruction on installation in both the book and on the disc.

**When will you get to see this work?**

Simple answer - no idea. It is a work in progress, it is a work being written in our respective spare time (all of us have full time jobs). Personally, I'd like to see the first couple of chapters finalised by mid October. Why so long though? Simple. We are taking the introductory chapters very slowly so that the reader will fully understand key concepts. This does have the inherent danger of going too slowly as to patronise the reader, so it must be guarded against.

# 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.*

*Note that 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.*

*Francis Glassborow*

### Student Code Critique 17: The Entries

*Last time I asked for a critique of the following short program.*

```
template <int N>
class T {
  public: friend T operator+(const T&,
                             const T&);
  private data[N + 1];
};
template <int N>
T<N> operator+(const T<N>& S1,
               const T<N>& S2) {
  return S1;
}
int main(){
  T<64> a, b, c = a + b;
}
```

*The critique was to include more than just identification and correction of errors. I wanted to see positive advice to the student on how better to achieve the objective.*

*When I set the problem I realised that it would actually be rather more demanding than some of the earlier critiques. That I was right in this is illustrated by the fact that I only received three submissions. It is a pity that only one in four hundred members can manage to submit a solution. Yes, I know about time, but this was summer for most of you. Perhaps lying around on sun washed beaches is more attractive. Interestingly the three entries all come from outside the UK (India, US and Norway) and those members have less time to enter than those in the UK.*

### From Gurusami Annamalai <d0253028@ncb.ernet.in>

#### 0. Need for a template parameter

The first question that needs to be asked is "Is there a need to make the class T a template?" This question is important because, when we make a class a template, and there are too many instances of the template, then the space complexity of the resultant object code would increase. Paraphrased, improper usage of template class would lead to code bloat.

There is a more fundamental issue here. For example, in the above program, if the template parameter, specifies only the attributes of objects, rather than their behaviour, then its better made a member variable of the class, and the value could be assigned in the constructor. This means that, if

```
T<25> t1;
T<50> t2;
```

behave the same (only their state differ), then the usage of templates is not justified.

I'll assume that class T being a template is justified and proceed further.

#### 1. Specifying the access modifiers

When we specify an access modifier, like `public:` etc, it doesn't apply to one member of the class. Rather it applies to all the members of the class that are declared after the modifier, either till the end of the class declaration, or the occurrence of another access modifier, whichever is earlier. Because of this semantics, it would be more self-descriptive if the access modifier is specified in a line by itself, followed by the members which it qualifies.

For example, it is better to write

```
public:
  friend T operator+(const T&, const T&);
```

than

```
public: friend T operator+(const T&,
                           const T&);
```

The latter somehow gives the feeling that it applies only to the member along which it appears; the former, is self documenting. It seems to say that the following members are public. As a bonus, the former style helps to avoid errors like the one in the program, (reproduced below)

```
private data[N + 1];  // data type missing
```

#### 2. Operator Overloading

While overloading an operator (like '+') for a user-defined data type, it is better kept as a member function of that class, unless the left operand of the operator being overloaded is of a different class. In that case, a friend function would be used.

In the above program we have

```
T operator+ (const T&, const T&);
                       // friend function
```

Since the left hand operand of the operator (the first parameter) is a reference to an object of the class, for which we are overloading the operator, this function is best made a member function of the relevant class (here it is T). So we should prefer the following in place of the above.

```
T operator+ (const T&);  // member function
```

Suppose we want to do something like

```
T<25> p;
int j;
T<25> r = j + p;
```

then we would need,

```
T<N> operator+(const int&, const T<N>&);
```

which cannot be converted to be used as a member of template class T. Only in such cases friend functions (to overload operators) are to be used.

The classic example to demonstrate this requirement is the overloading of `operator<<` to work with the `ostream` object `cout`. Of course, there are other examples, but I have given one.

#### 3. Making friends with template function

The declaration,

```
friend T operator+(const T&, const T&);
```

taking account of its context in the above program, is the same as

```
friend T<N> operator+(const T<N>&, const T<N>&);
```

This, in spite of our best intentions, doesn't make any friends. This is because we are looking for a non-template function with name "`operator+`", which doesn't exist. (g++ compiler, version 2.96, warned me of this!)

The above declaration should be

```
friend T<N> operator+<N>(const T<N>&,
                         const T<N>&);
```

which can be simplified as, again taking account of its context in the above program,

```
friend T operator+<N>(const T&, const T&);
```

and further to,

```
friend T operator+<>(const T&, const T&);
```

#### 4. Fixing the above program

By fixing the above program we get,

```
template<int N>
class T {
public:
  friend T operator+<> (const T&, const T&);
private:
  int data[N + 1];  // int or anything
};

template <int N>
T<N> operator+(const T<N>& S1, const T<N>&S2){
  return S1;
}
```

```
int main() {
  T<64> a, b, c = a + b;
}
```

### 5. The idiomatic way

The better way to write the above code is,

```
template<int N>
class T {
public:
  T operator+(const T&); // member function
private:
  int data[N + 1];
};
template<int N>
T<N> T<N>::operator+ (const T<N>& S1){
  return *this;
// first parameter in given program
}

int main() {
  T<64> a, b, c = a+b;
}
```

*Gurusami Annamalai*

*Note that Annamalai is a student in Bangalore. Until today, his was the only submission, which left me somewhat worried because I am not sure he has exactly pinned down the problem. Then the following arrived in my email:*

## From Christopher Currie <christopher@currie.com>

### Syntax Errors

Though the goal was to comment on C++ idioms, there are some syntax errors that will need to be corrected first, before the code can even be compiled. The original code, as printed, with line numbers added for clarity:

```
1   template <int N>
2   class T {
3   public: friend T operator+(const T&,
4                              const T&);
5   private data[N + 1];
6   };
7   template <int N>
8   T<N> operator+ (const T<N>& S1,
9                   const T<N>& S2) {
10    return S1;
11  }
12  int main() {
13    T<64> a, b, c = a + b;
14  }
```

The syntax errors are in the declaration of data; the omission of a colon after `private` and the lack of a data type. Access control keywords like `private` and `public` are not to be confused with type modifiers such as `const`. They are labels that affect the access of all the declarations that follow them. For this reason, it is good style to keep them on lines of their own:

```
1  template <int N>
2  class T {
3    public:
4      friend T operator+(const T&,
5                         const T&);
6    private:
7      data[N + 1];
8  };
```

Now we can clearly see the other problem with data, the lack of a data type. Legacy C compilers allowed identifiers to be implicitly `int` in the absence of a type declaration. Although some C++ compilers allow it, this is not valid C++.

```
7      int data[N + 1];
```

Now the code will compile, but it will not link. The reason is subtle, and the answer in this case covers many points of style that code borrows from textbook examples. As we clean up the style of the example, hopefully the answer will become clear.

### Class Names & Template Parameters

A major source of confusion in the class is the use of T as an identifier. T is (over)used in books on template programming, most commonly to represent the type of the template parameter. In this case, the template parameter is not a type, it is an integral value. This illustrates the importance of using parameter names and class names that represent the purpose of the class or parameter. In this example, there is not enough context to know the purpose of this class, so we'll use the name `FixedBuffer` for the class, and `length` for the template parameter.

```
template <size_t length>
class FixedBuffer {
public:
  friend FixedBuffer operator+(
      const FixedBuffer&, const FixedBuffer&);
private:
  int data[length + 1];
};
template <size_t length>
FixedBuffer<length> operator+(
        const FixedBuffer<length>& S1,
        const FixedBuffer<length>& S2){
  return S1;
}
int main() {
  FixedBuffer<64> a, b, c = a + b;
}
```

This looks better, but now that the obscuring type names are gone, you might notice that the two `operator+` functions look a little different...

### Friend Functions

From the C++ specification:

> "When a template is instantiated, the names of its friends are treated as if the specialization had been explicitly declared at its point of instantiation."

In our code, the function is treated as if it was declared as it appears in the friend function declaration. This draws attention to what the friend function is saying. As is common in template classes, the compiler does not require that you include the template parameters when using the class name within its declaration. The precise declaration would read:

```
template <size_t length>
class FixedBuffer {
public:
  friend FixedBuffer<length> operator+(
        const FixedBuffer<length>& S1,
        const FixedBuffer<length>& S1);
...
};
```

But since it is within a template declaration, the name lookup doesn't happen until the class template is instantiated. When our class declared in main, it's template declares a friend function that looks like:

```
FixedBuffer<64> operator+(
        const FixedBuffer<64>& S1,
        const FixedBuffer<64>& S1);
```

Aha! This is not a template function! This is why the program will not link, for the signature of this function does not match the template function that is defined later in the code. In this case, one correct declaration would be:

```
template <size_t length>
class FixedBuffer {
public:
  friend FixedBuffer operator+<length>(
                    const FixedBuffer& S1,
                    const FixedBuffer& S1);
      ...
};
```

(There is another syntax that my compiler accepts that requires that you declare the function before defining the class. I prefer the above.) This designates `operator+` as a template function, and shows that the concrete instance of the template function that has a matching length template parameter is the actual friend.

### Idiomatic usage

Wow, all these rules for template friends of template classes are hard! Fortunately, there is a better way. The important point to realize is that `operator+` doesn't have to be a `friend`! Think about the following:

```
int a, b, c;
// ... assign values
a = b + c;
```

Are `b` and `c` any different after the assignment than before? Of course not. The calculated result doesn't affect either of its parameters. In fact, we can't change either parameter, because its arguments are (as in this case) almost always `const`! The few operations that need rights to change an operand include the assignment operators and their kin, and these are almost always member functions that don't need to be friends anyway.

```
template <size_t length>
class FixedBuffer {
public:
  FixedBuffer& operator+=(
                      const FixedBuffer& rhs){
    // ...add rhs to *this
    return *this
  }
...
};
```

This code illustrates the canonical definition of the add-and-assign operator that takes its right hand argument as an operator, and returns a reference to itself, allowing the result to be used as an r-value (in other words, you can add the result to other variables, use it as a function argument, etc.). We've contained the code for adding `FixedBuffer` types within a member function, without any complicated `friend` functions.

Now, instead of duplicating that code in `operator+`, we'll simply reuse it in the canonical definition of addition:

```
template <size_t length>
FixedBuffer<length> operator+(
               const FixedBuffer<length>& S1,
               const FixefBuffer<length>& S2){
  FixedBuffer<length> temp(S1);
  temp += S2;
  return temp;
}
```

This very neatly allows us to change the definition of `operator+=`, and get an updated definition of `operator+` absolutely free. This is typically the way most of the arithmetical operators are defined for classes that need addition and subtraction to work the way that we expect that integers do. Remember this need not only apply to template classes; regular classes can benefit just as much from this technique.

### Summary

This deceptively small piece of code contained a lot of complexity, and in our analysis we can deduce a couple general rules. Remember, like all rules there are exceptions, but these will give the student place from which to build experience:

1 When a function needs to modify private or protected data within a class, make that function a member function if at all possible.
2 Define non-member, non-friend functions in terms of member functions to maximize maintainability and correctness.

*I was about to put this column to bed when the following arrived in my inbox.*

## From Terje Slettebo <tslettebo@chello.no>

I've been waiting for a C++ entry, and finally, one came. :)

Also, since I unwittingly "submitted" a blooper (which ended up as "Problem 3" in "Francis' Scribbles", C Vu June & August, :) ) let me try to "remedy" that by submitting this entry. By the way, in the following discussion, I agreed with Francis, about the problems with that code. It was originally intended to be just a test of a calculation, as well, not a well-tested routine. (*Indeed you did, it just made a good problem for my column because it gave me an excuse to write about something I wanted to write about anyway. I am like that, scavenge for bits wherever they may be found.*)

Regarding the given program. It says it can be compiled, but not linked. However, there are syntax error that makes it not even compile. These may be typos resulting from transferring the program to the magazine, though.

There are quite a few problems with this code, so let me take them in order:

1. Missing ":" after `private`.
2. Missing type of `data`, `int` chosen arbitrarily to fix it.
3. The `operator+()` doesn't make much sense, as it just returns one of the operands.
4. More seriously, the `friend` declaration declares a function, not a function template.

Point 4 leads to the following problem: When it comes to `c=a+b` it will have seen the following declarations:

```
T<64> operator+(const T<64> &,const T<64> &);
// friend declaration only, instantiated
// by the T<64>.
template<int N>
T<N> operator+(const T<N> &,const T<N> &);
// Declaration and definition
```

Since the first is an exact match for `c=a+b;`, it won't instantiate the template, and you get a link error.

The fix is easy enough: Make the `friend` declaration a `friend template` declaration. One also needs to provide a reasonable `operator+()` implementation:

```
template<int N>
class T {
public:
  template<int M>
  friend T<M> operator+(const T<M> &,
                        const T<M> &);
private:
  int data[N+1];
};
template<int N>
T<N> operator+(const T<N> &S1,const T<N> &S2){
  T<N> temp;
  for(int i=0;i!=N;++i)
    temp.data[i]=S1.data[i]+S2.data[i];
  return temp;
}
```

As another comment, in general, "T" doesn't appear to be a very meaningful class name, either, but it's hard to tell, without knowing the context.

OK. On to the alternative. The article said "I want suggestions for coding idioms that will make the student's life easier," so let's see what we can do. A common idiom is to implement an operator in terms of the assignment-version of the operator. See for example [1].

In the case above, `operator+()` could be a member function, but if you have an operator where the left-hand argument is another type, then it has to be a global function. So let's design a class for this general case, making the operator a global function. We'll also make a sensible `operator+=()` implementation, that `operator+()` will use. The compiler-generated copy constructor does the right thing, so none is provided, and other constructors are omitted, for brevity, and since they didn't appear in the original:

```
template<int N>
class T {
public:
  T &operator+=(const T &other){
    for(int i=0;i!=N;++i)
      data[i]+=other.data[i];
    return *this;
  }
private:
  int data[N+1];
};
template<int N>
T<N> operator+(const T<N> &S1,const T<N> &S2){
  return T<N>(S1)+=S2;
}
int main() {
  T<64> a,b,c=a+b;
}
```

This has a number of advantages over the first one:

1. By providing both += and +, you let the user of the class decide which to use, knowing that the former is generally more efficient, as it avoids creating a temporary object.
2. By implementing + in terms of +=, you ensure a consistent behaviour (and expected efficiency) for the two operators. The operation only needs to be implemented in +=, and you essentially get + "for free."
3. An important point is that you don't need any `friend` declaration, unlike the first version. `operator+()` needs no special access to T, as it only uses the public interface.
4. By implementing the `operator+()` as a general template, you may actually use it for several classes. Here, we only implement it for the class T, though.

Some details about the implementation:

- As given in [1], by using an unnamed temporary in the function template (rather than `T<N> temp(S1); S1+=S2; return temp;`), we make it possible for the compiler to perform the return value optimisation (eliminating the creation of any temporaries, by constructing the result at the call site), which may be available in more compilers, than the more recent named return value optimisation.
- It may be debated if `operator+=()` should return `T&` or `const T &`. However, the advice in [2] and precedence in the standard is to use `T&`, so that's what is used here. The rationale is basically to "do as the `int`s do."

#### References:


[1] Scott Meyers *More Effective C++*, "Item 22: Consider using `op=` instead of stand-alone `op`."

[2] Scott Meyers *Effective C++*, "Item 15: Have `operator=` return a reference to `*this`."


### The Winner of SCC 17

The editor's choice is Christopher Currie.

Please email `francis.glassborow@ntlworld.com` to arrange for your prize.

## Student Code Critique 18

*I have taken particular care not to introduce typos into this piece and I have quoted the first part of a twenty-line error message, not least because it is less than helpful. The specific problem is one that you either see straight away or that will take you an embarrassingly long time to identify. However there are a number of other points that should be spotted and commented on. Remember you should never allow your students to go away with no more than the instant fix they seek.*

I am trying to compile the following code and getting error that I don't understand.

In file `pgsimeta.h`

```
#include <vector>
namespace PGSIMeta {
class PgsiMeta {
public:
  PgsiMeta();
  virtual ~PgsiMeta();
  bool operator==(const PgsiMeta);
private:
// MetaData is a class defined at the top
  typedef vector<MetaData> DataList;
  DataList dataList;
};
}
```

In file `pgsimeta.cpp`

```
#include "pgsimeta.h"
using PGSIMeta;
bool PgsiMeta::operator==(
                  const PgsiMeta& obj){
  return dataList == obj.dataList;
}
```

Here is the error compiling:

```
_pgsi_meta.h:51: `PGSIMeta::operator== (const
PGSIMeta::PgsiMeta &)' must take exactly two
arguments
```

*Try to get entries to me by early November (by Nov. 17 at the latest)*

# The Wall

## Letters to the Editor

#### Regarding Changes to C Vu

Under the title "this->evolve()", James wrote about the scope of C Vu. I don't agree that C Vu shall be a magazine for Python or Java users. Firstly, if this happens, I would feel left out as a Perl user. No, C Vu cannot cover every language. Secondly, I don't think the world needs another forum for Python/Java/... If I want to get in depth with Python or Java, then I would turn to existing forums where there is enough depth and big audiences.

However I think it would be interesting for C Vu to include articles on other languages with a C/C++ slant, such as how to use C/C++ libraries from Python/Java and comparisons of languages in various aspects. Yes I expect a few articles would favour C++ over Java just as a Java magazine would favour Java but I don't see any problems with this.

Sorry for complaining when I am not contributing to C Vu. However I do enjoy reading C Vu even if there are non-C/C++ features like "Professionalism in Programming" and "A Short History of Character Sets".

Best regards


*Sven Rosvall*

`Sven_Rosvall@programmingresearch.ie`


*Thank you for taking the time to express an opinion. I certainly don't see C Vu as being a magazine for Python or Java users, but I do see that carrying a small amount of content for languages other than C and C++ is appropriate. I hope that the readership will make their opinions known and shape the form that C Vu takes in 2003. – James*

#### Some Pitfalls:

Dear James,

I received the following collection of traps hidden in typos and misconceptions from David Caabeiro `<dac@globalmente.com>` (one of our newer members). They are exactly the kind of thing that I was looking for when I presented the original little puzzle in my column. I wonder if other readers can be stimulated by this to add a few more of their own.

```
class foo {
public:
  static void f() {}
};
void f() {}

foo:f(); // this calls global f()

switch(c){
  case 0: return 0;
  case 1: return 1;
  defualt: return -1; // default misspelled
};

string s(); // This is a declaration, not a
            // construction with no args

int main(){
  int *v = new int(10); // instead of new
int[10]
  for (int i = 0; i < 10; ++i) v[i] = i;
}

char names[] = {
  "David",
  "John",
  "Peter" // comma forgotten here!
  "Mary",
  "George"
};
```

And finally the classic:

```
  void foo() {}
  foo; // Forgot()
```

I hate to think how many hours I have wasted with variants of that last one. Of course good compilers at a sufficiently high level of warnings give a diagnostic warning for many of them. That alone should be a good reason to switch to high warning levels. Unfortunately that often generates spurious warnings from third party libraries resulting in us fiddling around with `#pragma`s to hide them.


*Francis Glassborow*

`francis.glassborow@ntlworld.com`

James,

Some suggestions in response to "A Little String Thing" by Paul Whitehead (CVu 14.4).

1 The code can be shortened by replacing the definition of `struct is_space` with a call to `ptr_fun(isspace)` from the header `<functional>`.
2 The version of `isspace` Paul uses does not come from `<locale>`, but from the C-compatibility library `<cctype>`. To use the `<locale>` version you must pass a `locale` object as a second argument. This can make the utility more flexible.
3 I think it's good practice to use `const_iterator` whenever you don't need an writable iterator. It's safer.
4 I get suspicious when I see things like the following:

```
str = (str_start <= str_end) ?
      std::string(str_start, str_end) : "";
```

This checks for a "special case". Special cases are inelegant. Programs are meant to model the real world, but in my experience there are not as many special cases in the real world as programmers think there are. Is the above really a special case? I don't think so. The need for the test stems from scanning the whole string backwards. Since it has already been scanned forwards, and a starting position already found, we only need to scan backwards up to that position.

So we have:

```
void rem_space(std::string& str) {
  typedef std::string::const_iterator str_it;
  const str_it str_start =
     std::find_if(str.begin(), str.end(),
                  std::not1(ptr_fun(isspace));
  const str_it str_end =
     std::find_if(str.rbegin(),
        std::string::const_reverse_iterator(
                                  str_start),
        std::not1(ptr_fun(isspace)).base();
  str = std::string(str_start, str_end);
}
```

Personally, I would have preferred this function to return the result instead of changing the parameter:

```
const std::string trim(const std::string &);
```

Finally, it can be advantageous to templatize it so that it can work with `wchar_t` as well as `char`. I leave that as an exercise.

*Klitos Kyriacou*
klitos@klitos.org

*Paul Whitehead has provided a response to the message from Klitos, which I reproduce below:*

James,

I would like to reply, at least in brief, if only as a kind-of "thank you" for replying to the original article. Firstly I'd like to say thanks to Klitos Kyriacou for reading the article. Secondly, I'd like to say thanks for taking the time and effort to reply. It is appreciated.

Thirdly, I have an admission to make: I wrote the article/code in January this year (just after the heady days of Christmas/New Year celebrations :-). Since then, I've been changing compilers and platforms so often and have also had a hard-disk failure (yes, _of course_ I backed-up the hard-drive regularly, just not the things it turned out I needed ;-) that I have to confess I've "mislaid" the original code; the article too, as it happens, but then it's been reprinted in CVu so at least I can read my own article there! Now, I did send a copy of the code to be made available on the ACCU website but I haven't seen it on there in any recognisable form as yet, so I'll have to do some of this from memory - bear with me, if you can.

Points 1 and 4: yes, `ptr_fun` is good. Tidies things up a little. However, my own dissatisfaction with the solution I provided is that I find the whole thing just too verbose for what seems to be a rather trivial problem. Later on (point 4) you talk about a "special case" and say that it is inelegant. I would go further and say that special cases do not exist. They are simply a degenerate case of a more general rule - a general rule which just hasn't been found (or even looked for?) yet. When I do design reviews (and in general they tend to be OO and UML) I consistently weed out special cases as it shows, at least to my mind, that the problem is not correctly understood and therefore the solution is not correct. If this sounds a little extreme, then at least you can see that I fully support your point (4). I would, however, like to take the thought processes in your point (4) a lot further. As I have already mentioned, I find the whole thing too verbose and I suspect a more radical re-think of the initial solution - along the lines of your point (4), but more of it - may be required. Any takers amongst the C Vu readership?

Point 2 re: locale - yes, passing a `locale` to `isspace` does call up the locale version and this, as you rightly say, will make the utility more flexible.

Point 3 re: `const_iterator` - hmmm, well, yes, I do tend to use `const_iterator` whenever I can - and even "`const container_type::const_iterator it = ...`" where possible. I do this **despite** other people arguing convincingly that you may as well use the "simple" iterator in pretty much most cases. Scott Meyer's Effective STL springs to mind as having this guideline somewhere (but don't quote me, I need to check!). So using `const_iterator` is arguably not such a good thing as it may first appear. However, being stubbornly const correct (at least I hope I am!) I just can't bring myself to type "`iterator`" when "`const_iterator`" would work too.

Back to point (4) if I may. I'm not sure about passing back a const **copy** of the string object as it doesn't really get you much. A const ref, assuming, of course, it isn't a ref to a local (i.e. automatic) in the function or a const pointer (ditto) - that's fine. But a const copy? What are you going to do with it? If it is going to be used in another object then there's nothing stopping me doing this:

```
using std::string;
string mySpaceStr("   abc   def   ");
string myStr( trim(mySpaceStr) );
```

I could make `mySpaceStr` a const object and it still wouldn't change matters as `myStr` would still be non-const, as I wished - assuming I'm not just being sloppy and really did want a non-const `myStr` string object. Copy constructors (and assignment operators) in general take a const reference as their argument so by returning a const copy of the object from the trim function, all you are doing is passing that const object into a copy constructor in the above case) and then creating the non-const `myStr` object from it. A notable exception to the const argument for copy constructor/assignment operator is `std::auto_ptr<>` - and in that case it _may_ make sense to pass back a `const auto_ptr<>` as a return value, but that's a different story...

You mention making the function a template function. Yes, this would increase its flexibility somewhat. I would find the whole trim() function much tidier if it were to become a method on the `std::basic_string<>` class (thereby incurring the wrath of the multitude who believe `basic_string<>` has too many methods already!). If you consider the following, we could have:

1 `trim(string&)` - which modifies the argument, original proposal, or
2 `string trim(const string&)` - which doesn't modify the argument and returns a new string object (non-const, as per my comments above) as you suggest in your comments, or
3 method on the class. Sample usage: `string mySpaceStr( "  abc  def  "); mySpaceStr.trim();` After this operation, `mySpaceStr` now hopefully contains "`abc  def`"

I find option 3 preferable.

Regards,

*Paul Whitehead*

Dear Editor,

In a recent 'CVu' article (April 2002, Vol.14, #2, p.25), Francis Glassborow (FG) reviewed my latest book, "Embedded C". I am grateful to the editor of C Vu for giving me the opportunity to respond to this review.

Let me first say that I began writing software books almost a decade ago, and "Embedded C" is my third book: I have therefore been an author long enough to know that commenting on reviews is never a good idea. When your name appears on the cover of a book, you raise your head above the parapet. At this point, according to the unwritten rules of this 'game', the reviewers are allowed to take pot shots, and the authors are not allowed to fire back. In this case, given the tone and nature of FG's April 2002 article, I decided that I should break with tradition. Naturally, my comments will relate mainly to this review of "Embedded C". However, I will conclude by arguing that there are some more general lessons to be learned here for the ACCU as a whole.

FG opens his review by stating that "[Embedded C] is one of the most irritating books that I have reviewed for a long time". This theme - of 'irritation" - is maintained throughout the article.

FG's first source of irritation was, apparently, to find "too many places where text had been duplicated, almost word for word". As far as I am aware, the only text that is duplicated in this book is material in the preface, a small amount of which re-appears in later chapters. This is, I'm afraid, a fairly inevitable consequence of my writing style. I expect potential

readers to be able to skim the preface in a bookshop (or, increasingly, on line), and to come away with a clear understanding about the contents of the book. To meet this need, I create the first draft of the preface by assembling key sections of text from later chapters. In general, this approach works for me, and has never previously caused any comment from (let alone irritation to) a reviewer. In this particular book, the main body of the preface (excluding the 'Acknowledgements' section) is five pages long: even if all of this material appeared again, verbatim, somewhere in the 300 pages of the main body of the book, I doubt that the majority of readers would have noticed, let alone been irritated.

Despite his apparent concern about this issue, FG is only able to give one example of a piece of text which I have repeated. In this piece of text, FG has found an arithmetic error which I made when calculating the percentage of the microcontroller market occupied by devices from the 8051 family. FG is right: this is an error, and I should have picked it up. FG makes quite a lot of this error and - if it could be shown to be symptomatic of a 'sloppy' approach to my subject matter - I would understand his concern. However, this error is no more than a minor arithmetic slip-up that appears in an introductory paragraph and which has no bearing on the meaning of this paragraph, or on any other part of the book: if it is symptomatic of anything, it is only that the author is human.

The second thing that irritated FG was the source code layout. Here, his main criticism is that the font used for the source code listings is too large. Again, I agree, and hope that it will be possible to use a slightly smaller font in the next edition.

Another layout issue that concerned FG was the fact that in my code I use both "/* ... */" and "//" forms of comments. In my experience, this is not particularly unusual. FG clearly disagrees with this usage (as he is fully entitled to do), and he goes on to argue that - as a consequence of the font size used in the listings and my "inconsistent" comment style - the book suffers from "poor presentation of about 50% of the printed pages". In a room of 1000 ordinary developers (the intended audience of this review), I do not believe FG would find more than a handful that backed this assertion.

The third source of FG's irritation was the fact that I did not explain why the 'main' functions in my code examples do not return a value, and neither did I explain the 'unusual' declarative syntax used in Keil C when dealing with memory-mapped addresses. He's right: the book would have been improved if I had dealt with both of these issues.

The final main source of irritation for FG was my brief attempt to explain why C is a more appropriate language than C++ for use in embedded systems. This is an introductory book and my main point was very simple: accessing 'ordinary' variables in a C program carries less of an overhead than accessing private data encapsulated in a C++ object. I sought to illustrate this point with some simple code examples in C and C++. I do not think the point I was making was at all contentious. Apparently, however, my way of approaching this issue had FG "spitting" (his word). In his response, he makes a number of detailed observations about C99, new keywords and something called VLAs, and argues that the language used for embedded systems is not Standard C. I'm sure that FG's comments are all factually correct, but they have little to do with the - very simple - point I make in this introductory book about the overheads which result from O-O programming.

FG concludes his review at this point by suggesting that all is not lost, and he sees "scope for improvement" (my words) in a second edition: indeed, his closing sentence suggests that Addison-Wesley should get started on this next edition right away.

As I have made clear above, I agree with the great majority of the comments which FG makes in his review of "Embedded C", and they will be very useful when, in due course, we come to work on the second edition. Having said that, no book is perfect, and someone who wishes to find fault will always be able to do so. In this case, in my view, the reviewer has taken what most developers would view as minor issues and blown them up out of all proportion. This seems to me to be an over-reaction which might be expected (if not justified) in an academic journal: however, in a magazine aimed at professional developers it seems wholly inappropriate.

In addition, although it is far from clear from his review, FG's last detailed comment refers to material at around page 84 of "Embedded C". There are around 200 more pages. Should I conclude that they are free of irritation, or that FG gave up in disgust at this point? If the latter is the case, this would be a great pity, because most of the important material in this book appears after the introductory chapters which FG chose to focus on. For example, in Chapter 7, I describe how to create and use a very simple operating system for embedded applications and - by Chapter 10 - I discuss the creation of a complete embedded system (an intruder alarm system), which uses the operating system and other key techniques from the earlier chapters. Sadly, FG either did not read this material or, for some reason, did not feel it was worthy of a mention. This seems a pity, because I would have thought that people reading the review might have liked to be aware that this material existed.

Of course, the fact that I - as author of a book - disagree with a critical review is unsurprising. However, I think that this review may have some more general lessons for your organisation. In the last decade (or so), according to the ACCU WWW site, FG has reviewed well over 800 books for you. I haven't attempted to make a very accurate count, but it appears that this single individual is responsible for more than 50% of the published ACCU reviews. As an organisation, you should - I think - have two concerns. First, your book reviews are - as far as most members of the public are concerned - the most important thing you do. At present, you are allowing one individual to dominate your book-review section. This seems a very risky strategy for any organisation (to give just one example, what happens when this individual hangs up his reading glasses?). Secondly, you are clearly expecting rather a lot of any one individual, since - on average - FG must be reading (and reviewing) at least a book a week, every week of the year. The consequences are inevitable (and not hard to predict). In this case, I don't know if FG read all of "Embedded C", but I suspect that he did not: he is certainly on record for having "skimmed" (his word) at least one other book he has reviewed for you recently [C++ (Nitty Gritty), reviewed in "C Vu", February 2002, p.30]. Perhaps you need to keep a record of the key skills of your members (or people outside the organisation) and ask people to review books in their area, rather than simply having what appears to be a "first come, first served" policy (which seems to mean that FG reviews everything he wants, and the rest of you fight over the scraps)?

Let me end on a lighter note. The cover of 'C Vu' says "Written by Software Developers for Software Developers". I think that, if you are to continue to publish reviews like this, you should amend this to read "Written by Pedants for Pedants". You'll then be much closer to the mark.

***Michael J. Pont***
6 June, 2002

*This letter has been published to show that there is no conspiracy here at ACCU. Authors work hard in preparing books, and are understandably upset when reviews are unfavourable or even harsh. At Mr Pont's request, I have written a second review of "Embedded C", independently of the first. I shall therefore keep my response to Mr Pont's lengthy letter as short as possible, given the number of points he raises. In order to preserve the reputation of ACCU's book reviews, we must be open and responsive to criticism.*

*I too noticed the duplication of material. This may be a style choice of Mr Pont, but I don't feel that it makes a book more readable. In fact, I found my flow as I read the book (and I did read the entire book) to be broken by this repetition. Later text is not repetitive, however, though the code is; I lost count of how many times a busy wait time delay was printed.*

*The irritation at the "explanation" of why C is more suited to embedded development than C++ is most understandable, given that FG knows perfectly well that there is no overhead in accessing a private data member through an inlined accessor function with most modern compilers. Indeed, the various compilers I use for embedded development all optimise this away to produce exactly the same code that would be given by accessing a public data member directly as in C. There are reasons to prefer C in many embedded projects, but an inherent overhead in OO is not among them. The claims of the author that C++ is unsuitable for this reason suggest to me that he is not a C++ expert, which is reasonable, but in that case one must question why he must write about C++.*

*Any of you who are at all worried by thoughts that FG might indeed take the pick of the books for review should read what he has to say about obtaining books for review. I should mention that FG wrote that without any knowledge of the content of Mr Pont's letter. The pleasure of writing book reviews is open to all ACCU members, and many of us are grateful to those like Francis who give up their time to review truly dreadful texts such as "C++ (Nitty Gritty)".*

*For more specific information on the book, including an idea of who might find it worthwhile in spite of its flaws, my review appears in the book review section of this issue. — James*

# Francis' Scribbles

**by Francis Glassborow**

## Compatibility Issues

One of the hottest topics in the C and C++ communities is the issue of compatibility between the two languages. There are two extreme views:

- C should be a strict subset of C++
- C has nothing to do with C++

Most of us are somewhere between those two. I think we need to spend some time considering the issues and our own positions. I am not going to spend much time on the technical arguments relating to the first of the above because Bjarne Stroustrup has a well considered position being published by the C/C++ Users Journal. However I think it fair to try to remind people about why Bjarne Stroustrup's position would tend to that end of the spectrum.

Many languages are initially designed by a single person. Some then go on to become standardised. When that happens most language designers retire to the background, bite their lips and let others get on with changing their brainchildren. The only case I am aware of where this did not happen is C++. Bjarne Stroustrup continues to play a major part in its specification and evolution. He not only continues to have a major interest in the development of C++ but is a key player in it. I hope he will forgive me for saying that I think that means that he continues to have strong emotional as well as intellectual ties to it. He has a very definite view of what it is designed for and how it should evolve.

Now right at the beginning Bjarne Stroustrup made a practical (political) decision to build C++ on top of C, even when that meant the design decisions were not those that would have been made in a green fields development. There were excellent reasons why that decision was right. It took a language that was rapidly gaining popularity and that was an essential tool for Unix development and supplied a migration path to a much broader based language. However, I am also convinced that Bjarne Stroustrup designed C++ to eventually replace C. That was part of his vision. But let me be absolutely clear, it was not part of the vision of those responsible for the ongoing design of C. Ritchie had passed the batten on to J11 and eventually WG14. Those that took Bjarne Stroustrup's view that C++ was to replace C migrated to J16 & WG21 in the period 1989 to 1992. Those that believed C had an independent place in the World stuck with J11 & WG14. That is the root of trouble because there is a way in which both groups believed at some deep level that they were the guardians of the true flame.

One consequence has been the tendency for some C experts (aficionados) to decry C++ as a deeply flawed design that failed to learn the lessons of the past. At the same time some C++ enthusiasts were determined that C was an interesting historical relic that should have been quietly laid to rest. Meanwhile the vast majority of ordinary practitioners were receiving conflicting messages. Often introductory books had whole sections on 'C++ as a better C', and far too many writers got away with describing C++ as a superset of C. None of these people were ill-intentioned but the upshot is that we have a vast number of programmers who talk about something called C/C++ and believe that those who prevent C from being a subset of C++ are being obstructionist.

A major issue is that the two languages are close enough so that it is advantageous to some to be able to write code that will compile correctly with both a C and a C++ compiler. One key group are those responsible for the Standard libraries for C and C++. They do not want to have to write separate versions of the common parts of these libraries for the two languages. But writing versions that compile correctly for both languages is hard work, and they can see that a unification of the languages would reduce their work.

Then there are many developers working in C++ who want to be able to use libraries that have been written in pure C. This kind of compatibility makes good sense but even the common parts of the Standard libraries highlight serious issues with this approach. Look at `strchr()`. Its first parameter is a `char const *` in C, and it returns a `char *` based on that input parameter. The parameter has to be `const` qualified if it is to work with both mutable and immutable C-style strings. But the return type must **not** be `const` qualified if it is to be usable with mutable arrays of `char`. C has to take the perspective of trust the programmer because it does not have a sane alternative. C++ fixes the problem by splitting `strchr()` into a pair of overloaded functions. But this is not cost free, not only does it require overloading, but it also means that we have to forgo having a unique address for `strchr()`. In other words we cannot pass a function pointer to functions like `strchr()` in C++. Of course there are other solutions to designs where C would use a function pointer. My point is that even unifying the semantics of const for C and C++ results in a ripple effect, one that may not be acceptable to the community that uses C as its main development tool. And it is that community that J11 and WG14 is supposed to serve (how well they meet that obligation is a different issue). It is not the primary responsibility of those committees to consider the needs of programmers who want to write code that will compile identically in both C and C++. That does not mean that these committees have no responsibility to such people, which is why they attempt to remove gratuitous incompatibilities. However that is a far from easy task as even among those responsible for C++ most would be reluctant to claim they understood all the implications of the language design. How then should we expect those who are C specialists to understand the fine detail of C++.

Let me consider another aspect of unification, the spirit of compromise. It is my contention that that serves neither language. In a recent article in CUJ Bjarne Stroustrup reiterates a proposal to make the semantics of `void*` in C++ those that it has in C. In other words, remove the need for a cast to convert a `void*` into any other pointer type. In the spirit of compromise he is proposing that we weaken the C++ type system. From his perspective unification is more important than a design decision that has frequently been given as an example of the way in which C++ is better than C. Sorry, but I cannot buy that.

Of course if you start from the premise that C and C++ should be unified because it is an unfortunate error of history that they are not then compromise to get unification makes sense. And once they are unified efforts can be made to recover the high ground. But I think history teaches us something different. The widespread adoption of C++ probably did depend on a substantial degree of compatibility, though I am not so certain when I look at more recent experiences (e.g. Java and C#). It seems that languages that meet genuine commercial needs survive, and those that do not die. Algol 68 is arguably a much better designed language than Fortran but the world of numerical methods came down on the side of Fortran. The decision had nothing to do with compatibility. And we could ask how compatibility helped C gain such a strong foothold.

There is no other case where compatibility with an existing language has been an issue. Indeed the concept of compatibility only makes sense if C++ was intended to be a replacement for C in the same way that Fortran 77 was intended to replace Fortran 66. But if that were the case then it would have been the task of J11 and WG14 to standardise C++.

### Self-Compatibility

What almost every developer wants is to be able to use C++ libraries, and many also want to be able to use C ones as well. But being able to do that requires an entirely different set of requirements.

There are two major considerations. The first is that of an ABI, an applications binary interface (I hope I got the term right). This means that fundamental types and compatible user defined types (i.e., ones that can be declared in both C and C++, the so called POD types) have to be compatible. But this isn't even necessarily true between object code produced by different C implementations on the same platform and is completely unsupported when it comes to C++ (e.g., different implementations use different name mangling algorithms). Of course these are accepted as being non-standards issues, correctly so in my opinion.

Without a per-platform agreement on such things as the size and layout of fundamental types there is no compatibility and we have to resort to code being shipped as source, even if it is just plain C. And now we start to get reasons why the user might want to compile the same source code with both a C and a C++ compiler. Actually he does not, he just wants to be able to compile his entire source code with his C++ compiler. Establish good ABI standards on each platform and that problem goes away. If I do not need to compile C source because I can use a library shipped as object code then I largely stop being concerned about source code compatibility between C and C++. I could argue that increasing compatibility between C and C++ will make things worse because it will reduce the pressure for ABIs.

The second problem is much nastier, that of fundamental types. The core design of C means that new types such as `complex` just about have to be fundamental types. Providing a fully functional`complex` type in C without language provision for operator overloading is impossible. Note that operator overloading leads to reference types. But introducing such features in C leads almost inevitably to C++.

On the other hand the natural way to introduce new types into C++ is via appropriate libraries. I think that the designers of C++ would be very loath to introduce a bundle of new fundamental types just for the purposes of compatibility with C. C++ provides powerful tools for making user defined types first class citizens. That is one of the strengths of C++ and it would be odd to revert to fundamental types whenever C decides that it needs a new,

fully supported type. Of course we can argue the merits of C having a complex type, but I contend that that is up to the pure C community to determine.

I do not know how we solve the problem of link time compatibility between a fundamental type in C and a user-defined equivalent in C++. However if we focus our efforts on link time compatibility we might solve the problem. I am certain that if we pursue the path of unification of C with C++ we will not have the energy and resources to deal with the real and immediate problems.

### Food for Thought

Have a look and see how many new books for C novices have been written in the last decade. Yes, one consequence of stability was a marked reduction in publishing interest. Now look and see how many new books have been published for C++ novices in the last couple of years. Some argue that the visibility of such books indicates the health of a language. If that is true, C is nearly dead and C++ is definitely on the way out.

C++ programmers know this is not true, so why do they think it is true of C?

Just as a matter of interest, how many different compilers for pure C do you think there are on the market today? Of course most of those are not in the shrink-wrapped market because those that need them know where to go and will want to buy multiple licenses.

Next time someone asserts that C is dead, or that C++ is dying, ask for their evidence. And when they give it, check its relevance because mostly such statements are based on a pretty superficial perspective.

## Problem 4

When doing code inspections you need to cultivate a suspicious mind. In that light consider the following simple function and comment on what you would check and what minimal changes you would require.

```
void foo() {
  mytype* mt_ptr = new mytype;
  bar(mt_ptr);
  delete mt_ptr;
}
```

I wonder how quickly you realised that it was essential to look at the definition of `bar`. Once you get the idea, the longer you think about the disasters it can perpetrate the worse it gets. For example, which of the following function declarations is the greatest harbinger of doom:

```
something bar(mytype *);
something bar(mytype const *);
something bar(mytype * const);
something bar(mytype const * const);

something bar(mytype * &);
something bar(mytype const * &);
something bar(mytype * const &);
something bar(mytype const * const &);
```

The first instinct that many have is that the pass by value cases must be safe because they do not change the original. That instinct really needs quick modification. Not being able to change the original just might be the biggest disaster of all. Consider a pretty minimalist definition:

```
mytype * bar(mytype* mt_ptr) {
  delete mt_ptr;
  return 0;
}
```

Now you see it. Just because a function gets a copy of a pointer does not mean that it cannot damage the original because it can simply invalidate it. Of course the function is pure madness, particularly with that name, but there are lots of mad programmers out there. Oh, and in case you were wondering, it does not matter how many `const` qualifiers you apply to the parameter, you can still apply `delete` to it in Standard C++.

So what about the cases where the pointer is passed by reference? Well have a look at:

```
void bar(mytype * & mt_ptr_ref) {
  delete mt_ptr_ref;
  mt_ptr_ref = new mytype[1];
}
```

See the problem this time? Exactly, your change requires that `delete[]` be called on the `mt_ptr in foo()`.

The nasty thing about the problem that I am trying to highlight is that looking at the function declaration tells you very little of use. You have to look at the definition. Pointers are like scalpels; they are highly refined tools that experts can use constructively. However they are lethal in the hands of others. Because C++ is designed as a language to meet the refined purposes of library designers as well as the coarser needs of the application programmer we have to learn the danger signals:

1) Raw `delete/delete[]`
   Anytime you see a `delete` or `delete[]` in a free (i.e. non-member) function be deeply suspicious. If you see one in a member function, have a look at the destructor for the type. We should only be destroying what we own. Sometimes we delete one thing to replace it with another, but that should only be done inside a type that owns the resource.
2) Suspect raw pointer parameters
   There are many cases where these are fine, but you should have checked the quality of the programming that produced these functions.
3) Do not use raw pointers for dynamic resources.
   I think that one is close to an absolute injunction. It is the task of a destructor to release resources. Dynamic resources should be owned by something that releases them in a destructor. Generally this ownership should be provided by some form of smart pointer.
4) Do not mix arrays with single instances
   Sometimes we create single instances dynamically. `auto_ptr` is designed to handle the lifetime of such objects. It has slightly unusual semantics, with the result that it should always (well almost) be passed by reference and returned by value. Think carefully till you understand why the ownership semantics of `auto_ptr` lead to that coding guideline. There are other smart pointers that have more sophisticated semantics, however they are almost all designed for single objects. When it comes to arrays we should distinguish between those whose size is fixed and those whose size can vary. Mostly, where the size is fixed we should consider a simple raw array. Its semantics will almost certainly meet our needs. If we need variable size, or there is some other reason that we need to use dynamic memory for our array then our prime candidate should be a vector. We should not be calling `new[]` ourselves. The STL `vector` encapsulates a dynamic array so that all the handling is done for us. If we need the actual array (for example, to pass to a C function that takes a pointer to an array) we can extract the address of the internal array. That address remains valid until you do something that causes the vector to relocate its internal storage. I know that this is not guaranteed in the original C++ Standard, but it should have been and a response to a defect report has corrected that omission.

In conclusion: Most programmers should never use `delete` and even fewer should use `new[]` and `delete[]`. Dynamic resources should always be owned, usually by some form of smart pointer or container. You should not be using `delete` on parameters, though `delete` applied to member that is a pointer can sometimes be acceptable. Note that this is a stronger statement than the commonly made request that the language should not allow `delete` on a `const` pointer or pointer to `const`.

As I have said before, cultivate a suspicious mind and learn to program responsibly. Just as you have to trust other programmers, make sure your own code merits the trust of others.

## Problem 5

Consider the following brief program. What is the output and why?

```
struct base {
  virtual void report {
    std::cout<< "base" << std::endl;
  }
};

struct derived: public base {
virtual void report {
  std::cout<< "derived" << std::endl;
};

int main() {
  base * x = new derived;
  try {
    throw *x;
  }
  catch(base & br) {
    br.report();
  }
  return 0;
}
```

# Features

## What is a Hash Table?

**Victoria Catterson** <vic@cowlet.org>

A hash table is a data structure used for associating keys with values, with the goal of efficient storage space and fast access. Each item has a key, which the table's hash function will use to calculate the "hash value" for that item. The hash value indicates where in the table the item should be. The use of the hash function means that indexing into the table can be performed in constant time, regardless of the number of items in the table. A key can be of any type, but a hash value will always be an integer, as it represents the item's position in the table.

A hash table where each key results in a unique hash value is called directly accessible. This is the "ideal" hash table, because searching for items will always take O(1) time. However, in most cases this type of hash table is not practical, as it tends to be wasteful of memory. More common is the chained hash table, where many keys will result in the same hash value.

A chained hash table can be organised as an array of linked lists. The array positions correspond to the hash values. The linked lists are called buckets, and store items with keys that all translate to the same hash value. When a bucket contains more than one item, it is known as a collision.

As an example, a chained hash table is used below to store city information. A structure is defined to hold the city's name as a string, and the population in thousands as an integer. The pointer to another city structure is used for chaining cities in the one bucket. A very simple hash function has been chosen, where the key is the city's name, and the hash value is the length of the name. The buckets are not sorted, so each item is simply added at the start of the linked list.

The `print_table` function prints the name and population of every city in the table, one bucket at a time. Five cities are added to the table, and then `print_table` is called. The output of the program is:

```
Hash Table Contents:
Bucket 0: End.
Bucket 1: End.
Bucket 2: End.
Bucket 3: End.
Bucket 4: End.
Bucket 5: End.
Bucket 6: Dundee, population 147000; End.
Bucket 7: Glasgow, population 607000; End.
Bucket 8: Aberdeen, population 190000; End.
Bucket 9: Inverness, population 41000; Edinburgh,
                                    population 402000; End.
Bucket 10: End.
Bucket 11: End.
Bucket 12: End.
Bucket 13: End.
Bucket 14: End.
```

This is a perfectly valid hash table, but it is not a very good one. Even if more cities are added, the bucket lengths will not be balanced, with most city names colliding in buckets 6 to 9. In addition, buckets 0 to 2 are not likely to contain any cities! This sort of behaviour is poor for two reasons: buckets 0 to 2 are a waste of memory; and searching will be closer to O(n) than the O(1) ideal, because the concentration of items in 4 buckets makes the searching behaviour more like that of a linked list than a hash table.

This problem can be solved by choosing a hash function that will give a more even spread of items between buckets. In addition, it may be advisable to change the number of buckets. If a good hash function is selected that results in a relatively even distribution of items in buckets, then the number of items expected in each bucket will be roughly n/m, where n is the number of items in the table, and m is the number of buckets. If this is a realtively large number, increasing the number of buckets is advisable. In order to help achieve an even spread, m should not be a power of 2. A prime number is an ideal value for m.

As this example only contains five cities, a good value for m is 5. With a good hash function, the expected distribution will be one city in each bucket. However, it is tolerable to have one bucket with two cities, and one with none, as a hash function will never (well, rarely) be perfect.

The most difficult aspect of creating a hash table is deciding on the hash function. Very often the simplest functions will produce extremely uneven results, as in the example above. If the key is not an integer, it is a good idea to massage it in some way, so it can be treated as an integer. This technique was used in the previous example, where the key was a string, but the length of the string was used to produce the hash value. However, to produce better results something more random than the string length must be used to calculate hash values.

A relatively simple method is to add up the ASCII values of every character in the city's name, dividing by a prime number after each addition. This algorithm may not produce good results if the keys are very similar. For example, hashing "malloc", "calloc", and "realloc" with the new function produces very poor results, with all three hashing to the same value. However, there is no obvious pattern in the city names, and the function produces reasonable results, with only one collision.

One final stage must be performed before the integer is a valid hash value. As there are five buckets, the value must be taken modulo 5. This ensures that the hash value is a bucket number. The new hash function looks like:

```
int hash_func(struct city *item) {
  int sum, i;
  for( i = 0, sum = 0;
       i < strlen (item->name);
       ++i)
    sum = (sum + item->name[i]) / 3;
  return (sum % NUM_BUCKETS);
}
```

The new table looks like:

```
Hash Table Contents:
Bucket 0: Dundee, population 147000; End.
Bucket 1: Inverness, population 41000; End.
Bucket 2: Edinburgh, population 402000; Glasgow,
                                    population 607000; End.
Bucket 3: Aberdeen, population 190000; End.
Bucket 4: End.
```

And, of course, `NUM_BUCKETS` is now defined as 5, instead of 15.

It should be noted that the example hash table is efficient in terms of memory and indexing speed, but it is not very scalable. If the number of cities were to increase to 25, the time to find an item would be relatively poor. The average number of items in a bucket would be 5, and consequently the search time will tend more towards O(n) than O(1).

Generally, if it is known roughly how many items the table will accommodate, the number of buckets should be tailored to maximise efficiency, as in the given example. However, if the number of items is not known, or may vary greatly, a trade-off should be considered between indexing time and memory usage. More buckets can result in wasted memory, but faster indexing time.

Increasing the number of cities to 10 produces the following results:

```
Hash Table Contents:
Bucket 0: Havana, population 2241000; Paris,
                                    population 9469000; End.
Bucket 1: Bombay, population 15093000; Tokyo,
                                     population 26836000; End.
Bucket 2: Moscow, population 9233000; Beijing,
                                    population 12362000; End.
Bucket 3: Johannesburg, population 1849000; Perth,
                                    population 1220000; End.
Bucket 4: San Francisco, population 3866000; Lima,
                                    population 7452000; End.
```

This is the optimal number of items per bucket. Increasing the number of buckets to 9 gives good results, also:

```
Hash Table Contents:
Bucket 0: San Francisco, population 3866000; End.
Bucket 1: Paris, population 9469000; End.
Bucket 2: Bombay, population 15093000; Tokyo,
                                population 26836000; End.
Bucket 3: Moscow, population 9233000; End.
Bucket 4: Lima, population 7452000; End.
Bucket 5: Havana, population 2241000; End.
Bucket 6: End.
Bucket 7: Beijing, population 12362000; End.
Bucket 8: Johannesburg, population 1849000; Perth,
                                population 1220000; End.
```

This shows that the hash function chosen appears to be scalable, whether the number of buckets is increased or not. It can therefore be said that it is a good hash function for these data sets.

Another type of hash table is the Open Addressed hash table. This stores all items in the table array itself; items are never chained. When the hash function results in a collision, an algorithm is used to suggest another bucket. This algorithm is used repeatedly until an empty bucket is found, or all possible buckets have been inspected. This process of trying different buckets is known as probing the table.

The simplest open addressed hash table uses linear probing. This involves probing buckets in numeric order until an empty one is found. The bucket indicated by the hash function is the starting point, and each bucket is inspected in order until the collision is resolved.

Another probing technique is called double hashing. This involves two hash functions: the first to determine the starting bucket; and the second to determine the "stride". The stride is the value added to the bucket number to determine the next bucket to inspect when a collision occurs. [1]

A disadvantage of this sort of table is that the maximum number of items to be inserted must be known in advance. While the buckets of the chained hash table can be increased in size to accommodate items until there is no more memory available, the open addressed hash table's size is always fixed.

An advantage of the open addressed hash table is that the worst case search time is better than that of the chained hash table. In the first table, the worst performance occurs when the hash table is full, but the required item is not in the table. In this situation, searching takes O(m) time, where m is the number of buckets in the table. In the chained table, the worst performance occurs when all items are in one bucket, but the required item is not in the table. In this case, searching takes O(n) time. If n>m, then the open addressed table has the best performance.

Hash tables may seem rather daunting and complex, but this is only because of the many decisions that must be made. Once the type of table, number of buckets, and hash function have been chosen, implementation is quite straight forward. Next time a data structure is required, consider whether a hash table would be effective; it is surprising how often they can be used.

*Victoria Catterson*

## References


[1] For more information on double hashing and other probing algorithms, see http://www.cise.ufl.edu/~sahni/dsaaj/enrich/c11/overflow.htm

[2] Scottish city population statistics: 1991 Census, http://www.siliconglen.com/scotfaq/1_6.html

[3] World city population statistics: The Times Atlas of the World, 2001


```
/* hash.c by Victoria Catterson */
#include <stdio.h>
#include <string.h>
#define MAX_NAME_LEN 15
#define NUM_BUCKETS 15

struct city {
  char name[MAX_NAME_LEN];
  int population; /* in thousands */
  struct city *next;
};
/* This is the array of linked lists;
   the hash table itself. */
struct city buckets[NUM_BUCKETS];

int hash_func(struct city *item) {
  return (strlen (item->name));
}
void add_item(struct city *head,
              struct city *item) {
  item->next = head->next;
  head->next = item;
}
void print_table(void) {
  int i;
  struct city *current;
  printf("Hash Table Contents:\n");
  for(i = 0; i < NUM_BUCKETS; ++i) {
    current = buckets[i].next;
    printf("Bucket %d: ", i);
    while(current != NULL) {
      printf("%s, population %d000;",
             current->name, current->population);
      current = current->next;
    }
    printf("End.\n");
  }
  fflush(stdout);
}

int main (void) {
  struct city glasgow, edinburgh, aberdeen,
              dundee, inverness;
  int hash_val;

  strncpy(glasgow.name, "Glasgow",
          MAX_NAME_LEN);
  glasgow.population = 607;
  hash_val = hash_func(&glasgow);
  add_item(&buckets[hash_val], &glasgow);

  strncpy(edinburgh.name, "Edinburgh",
          MAX_NAME_LEN);
  edinburgh.population = 402;
  hash_val = hash_func(&edinburgh);
  add_item(&buckets[hash_val], &edinburgh);

  strncpy(aberdeen.name, "Aberdeen",
          MAX_NAME_LEN);
  aberdeen.population = 190;
  hash_val = hash_func(&aberdeen);
  add_item(&buckets[hash_val], &aberdeen);

  strncpy(dundee.name, "Dundee",
          MAX_NAME_LEN);
  dundee.population = 147;
  hash_val = hash_func(&dundee);
  add_item(&buckets[hash_val], &dundee);

  strncpy(inverness.name, "Inverness",
          MAX_NAME_LEN);
  inverness.population = 41;
  hash_val = hash_func(&inverness);
  add_item(&buckets[hash_val], &inverness);

  print_table();
  return 0;
}
```

# Enlarging on "A Problem of Access" in CVu December 2001 Vol. 13 No. 6

**Atul Khot**

We are developing a component based system where one central component is the server.

Among other things, the server holds some very large containers of objects (thus these containers are preferably constructed only once — at the time the server component is coming up.) A number of client components can refer to the container object simultaneously. They may inspect it (**shared** mode — container and its objects are read only) when there are no **exclusive** (**mutating**) mode clients, but if any client wants to change any such containers or objects within, it has to have an exclusive lock on the container first (**exclusive** or **mutating** mode ) and there must be no pre-existing **exclusive** or **shared** mode clients. This way, there is no (meaning "minimum"!) space for surprises.

One such container is `Dimension`, which holds a great many members (millions).

The following abstract class is exposed to outside clients (other components):

```
// Interface class
class CDimension_I {
  public:
    virtual Id GetId() = 0;
    virtual void setId(Id id) = 0;
    // other interface methods
};
// This class holds millions of members
// organized in a tree structure
class CBaseDimension
  : public CDimension_I, public Resource_m {
  public:
    Id GetId();
    void setId(Id id);
    //other helpers and concrete implementations
};
```

Client contexts are stored in objects of class `Session`.

```
CBaseDimension& cbd = ...;
Session* pSession = ...;
// ...
ResLock_m* pResLock = new ResLock_m(
    dynamic_cast<Resource_m*>(&cbd), pSession,
    LT_SHARED); //locks in shared mode
// inspect the container, traverse the
// data structures it holds etc.
delete pResLock; // unlock the above locked
                 // resource
```

A number of issues rear their ugly heads:

1 How can we handle locking transparently? The business logic gets muddled if we put locking calls in between.
2 How do we make sure that every Resource is locked, used and released? (i.e., no resource leaks).
3 What about multiple locking of the same resource by the same client?

Okay, let us take them in turn. First some definitions:

```
enum LOCK_TYPE { LT_NOLOCK,
                 LT_EXCLUSIVE,
                 LT_SHARED
};
template<LOCK_TYPE lckType>
struct LockTraits {};

template<>
struct LockTraits<LT_SHARED> {};

template<>
struct LockTraits<LT_EXCLUSIVE> {};
```

This is similar to the `Int2Type` template explained in "Modern C++ Design" by Andrei Alexandrescu. We essentially convert an enum into a type. Converting enums to types helps the C++ compiler track bugs for you. This is immensely helpful as you shall soon see.

Next, separate all the methods of the shared class into `const` and non-`const` methods.

Whenever, a client code asks about SHARED access, give a const reference to the shared object.

Only when it requests an EXCLUSIVE access, then only give out non-const reference, essentially allowing client code to call any method.

A singleton object of type `CGetResource` grants these accesses.

```
class CDimension_I {
  public:
    virtual Id GetId() const = 0;
              // note the const
    virtual void setId(Id dimId);
              // note the absence of const
    // other interface methods
};

const CDimension_I* inspectDim = ...;
CDimension_I* mutateDim = ...;

inspectDim->GetId();  // ok
inspectDim->setId(3); // compilation error,
                      // can't call a non-const
                      // method on a const object
mutateDim->GetId();   // ok, a non-const can
                      // call a const method
mutateDim->setId(3);  // ok
```

For more in-depth treatment of this pattern, please see a very neat article by Kevlin Henney in CUJ — "C++ Experts Forum" online article "From Mechanism to Method: Further Qualifications".

```
template<class DimT, class LckType>
class CdimWrapper {
  protected:
    DimT* m_pDim;
    CSessionBase* m_pSession;
  public:
    CDimWrapper(DimT* pDim,
                CSessionBase* pSession,
                LockTraits<LT_SHARED>)
        : m_pDim(pDim),
          m_pSession(pSession) {
      // Pass the buck onto session class.
      // This takes care of multiple locking
      m_pSession->Lock(
         dynamic_cast<Resource_m*>(pDim),
         LockTraits<LT_SHARED>());
    }
};

class CBaseDimShr
    : public CDimWrapper<const CBaseDimension,
                         LockTraits<LT_SHARED> > {
  public:
    CBaseDimShr(const CBaseDimension* pDim,
                CSessionBase* pSession,
                CMapShr* pMap,
                LockTraits<LT_SHARED>)
       : CDimWrapper<const CBaseDimension,
             LockTraits<LT_SHARED> >(
                  pDim,
                  pSession,
                  pMap,
                  LockTraits<LT_SHARED>()) {}
    const CBaseDimension* operator->() const {
      return m_pDim;
    }
};
```

```
class PtrBaseDimShr {
    const CBaseDimShr* m_pLwDim;
                        // note the const
    // private copy ctor and assignment op
    // not shown */
  public:
    PtrBaseDimShr(const CBaseDimShr* pLwDim)
       : m_pLwDim(pLwDim) {
    }
    bool operator!() {
      return m_pLwDim == 0;
    }
    operator const void*() const {
      return m_pLwDim;
    }
    const CBaseDimShr& operator->() const {
      return *m_pLwDim;
    }
    ~PtrBaseDimShr() {
      delete m_pLwDim;
    }
};

class CBaseDimEx : public
     CDimWrapper<CBaseDimension,
                 LockTraits<LT_EXCLUSIVE> > {
  public:
    CBaseDimEx(CBaseDimension* pDim,
               CSessionBase* pSession,
               CMapEx* pMap,
               LockTraits<LT_EXCLUSIVE>)
      : CDimWrapper<CBaseDimension,
            LockTraits<LT_EXCLUSIVE> >(
                pDim,
                pSession,
                pMap,
                LockTraits<LT_EXCLUSIVE>()) {
    }
    CBaseDimension* operator->() {
      return m_pDim;
    }
};

class PtrBaseDimEx {
    CBaseDimEx* m_pLwDim;
    // private copy ctor and assignment op
    // not shown
  public:
    PtrBaseDimEx(CBaseDimEx* pLwDim)
      : m_pLwDim( pLwDim) {
    }
    bool operator!() {
      return m_pLwDim == 0;
    }
    operator void*() {
      return m_pLwDim;
    }
    CBaseDimEx& operator->() {
      return *m_pLwDim;
    }
    ~PtrBaseDimEx() {
      delete m_pLwDim;
    }
};

class CGetResource {
// This class takes care of point 3
// (allow multiple SHARED locks or a single
// EXCLUSIVE)
  const CBaseDimShr* GetBaseDim(
           LockTraits<LT_SHARED>,
           BaseDimHandle hDimHdl,
           CSessionBase *session_ptr);
  CBaseDimEx* GetBaseDim(
           LockTraits<LT_EXCLUSIVE>,
           BaseDimHandle hDimHdl,
           CSessionBase *session_ptr);
  .........
};
```

Now a client acquires the shared object as follows:

```
void ClientClass::getBaseDimName(
                    CSessionBase *pSession,
                    CGetResource* pGetRes,
                    BaseDimHandle hDim ) {
  // original code
  // CBaseDimension* pDim = .....
  PtrBaseDimShr pDim =
         pGetRes->GetBaseDim(
            LockTraits<LT_SHARED>(),
            hDim, pSession);
}
```

When the above function exits (either with a return statement or by throwing an exception), this scheme guarantees that all locks are released. Also, all the locking mechanism is neatly hidden. The programmer just has to know

1 He is trying to get a shared resource ( `LockTraits<>` ) and
2 The call might sleep

Now with all this in place, suppose somebody does the following:

```
PtrBaseDimShr pDim = pGetRes->GetBaseDim(
                   LockTraits<LT_EXCLUSIVE>(),
                   hDim,
                   pSession);
```

As `LockTraits<LT_EXCLUSIVE>` is a class (an empty class), there is no function that satisfies the above line. The compiler will kick in an error.

Now, the programmer corrects the mistake and obtains a shared pointer.

```
Id id = 4;
pDim->setId(id); // this again will kick in a
                 // compiler error.
```

Let us trace this call to see what is going on. Trying to resolve the above call, as `pDim` is not a plain pointer, the compiler applies the `operator ->` found in class `PtrBaseDimShr`, which returns a const object of "`CBaseDimShr`". Reapplying the `operator ->` again, which returns a pointer to const object of type `CBaseDimension*`. So in essence, the above `setId()` call is invoked on a `const` object of type `CBaseDimension`. However, as this is a non-`const` method, the call fails to compile.

Similarly, tracing through an exclusive mode pointer, we can indeed see that the above call goes through fine.

This design was grafted onto a legacy code, so we did many other nice things so it all worked quite well. The only other thing that needed to be changed was the declaration of the pointers. (`PtrBaseDimShr` instead of `CBaseDimension*` etc.). As the code was quite huge, nearly 20000 lines and still growing, we never needed to know the logic so as to know where to place the lock calls etc. We just mechanically replaced the above declarations (using nifty VIM macros) and the work got reduced to just few keystrokes. The unit testing of the locking scheme was done using a Perl script (which is an altogether different story — will do it some other time).

One queer thing I notice again and again. After all the core design was coded in place, it was just a mechanical/routine matter to extend it. The system just developed and extended itself. I remember a similar experience while developing a schema language using Lex, Bison and C++. This way of things is indeed very gratifying. We did complete our part well before 15 days of the release deadline, leaving everybody happy and some more spare time for myself to linux it away.

*Atul Khot*

# XML Parsing with the Document Object Model

**David Nash**

## What is the DOM?

Following Tim Pushman's article on parsing XML using SAX, I will describe here the principles and details of the Document Object Model, which defines a standard way to model an XML or HTML object. It describes data structures with standard names and behaviours, and standard functions to access the data. Most XML parsers support the DOM and will parse any well-formed XML document into a DOM structure for you.

But first a bit of history... Long ago when web pages were mainly static text with a few images here and there, the writers of the two main browsers (Netscape and Internet Explorer of course) came up with what they called Dynamic HTML, or DHTML for short. DHTML allowed web pages to access their own content, and to change it according to users' actions. This allowed people to write web pages with images that changed when the user moved the mouse over them, and other fancy effects that they thought would attract more people to their web sites.

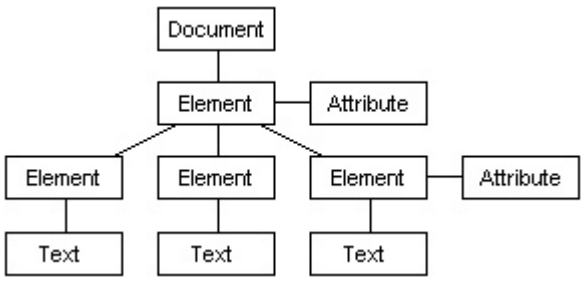
Both browsers achieved this by adding a scripting to capability to the dialect of HTML that they understood. The two scripting languages were similar but differed in many ways. They were called Javascript and Jscript. Neither is remotely related to Java and the two have now been unified and standardised into ECMAScript [1].

In order that a script embedded in an HTML document could have something to work on, a model was needed, through which the script could access the various elements on the web page. The model that was created modelled the actual HTML document itself, so became known as the Document Object Model, or DOM for short. The DOM is now a W3C standard [2] and comes in two slightly different alternatives, for XML or HTML respectively. Although, as we have seen, the DOM had its origins in HTML, the HTML version can be thought of as a slightly specialised version of the XML DOM, and since this is a series of articles on XML we will concentrate on that one here. You can read more about the HTML DOM in the official W3C recommendation [3].

The model described by the DOM can be thought of as a tree-like structure. The tree is made up of a number of different object-like nodes. A node can be any one of a number of different node types, which are effectively data types derived from the basic Node type. In fact the standard does not specify that nodes in the DOM have to be implemented as objects at all, merely that they behave like objects. This allows DOM implementations in non object-oriented languages like C.

In order to get an XML document into a DOM tree you need an XML parser that supports the DOM (most do). You can then parse existing XML documents and create a DOM structure, or create one from scratch. In this article we shall see how to do this, concentrating mostly on the Apache Xerces C++ parser, although the principles apply generally.

The root node of the tree is the document itself. Since this is a tree structure there can only be one root, and that ties in with the concept of an XML document, which can have only one root-level element (the "document element"). The other main types of node in the tree are Elements, Attributes, and text. In keeping with well-formed XML, the document node contains only one element, while elements can contain other elements, attributes, or text:

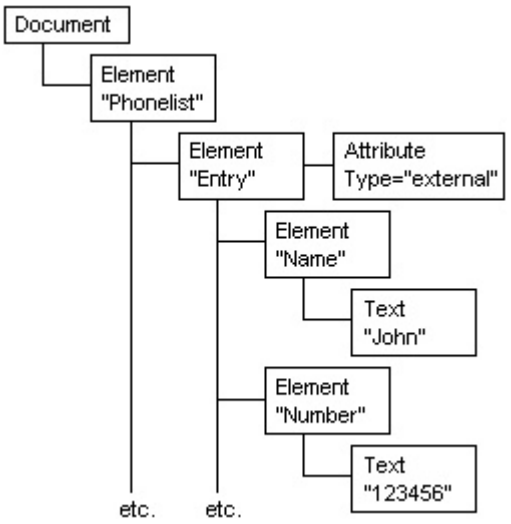



This is a good place to explain that, although the DOM defines all these types of node, you don't have to use them. There are actually two interfaces to a document via the DOM: through generic "node" objects (which the W3C recommendation describes as "the primary datatype for the Document Object Model") or through more concrete derived types "Element" objects, "Attribute" objects, and so on. This means that all the different node types in the diagram could be labelled "node", and accessed through the DOM Node interface. Each DOM Node has an attribute (called NodeType) that indicates what kind of node it actually is.

Let's look at an example. Take the following piece of XML (yes, it's the familiar hypothetical phone book!):

```
<?xml version='1.0'?>
<PhoneList>
  <Entry type="external">
    <name>John</name>
    <number>123456</number>
  </Entry>
  <Entry type="external">
    <name>Jane</name>
    <number>7654321</number>
  </Entry>
  <Entry type="internal">
    <name>Fred</name>
    <number>100</number>
  </Entry>
</PhoneList>
```

The DOM structure of this document would look something like this:




Once parsed into a DOM tree (we will see how to do this in a minute) you can access the XML document using the DOM Document interface. This has a documentElement attribute to access the single document element (`PhoneList` in this case). As an example of the methods provided by the DOM, you could find its child elements with a specific name using the `GetElementsByTagName` method.

## Parsing

Different XML parsers and processors have different ways to initiate the parse process that builds the DOM tree. The examples shown here all use the *Xerces* parser [4] from Apache. This parser is closely related to the IBM XML4C parser (IBM donated an early version to Apache, and their subsequent versions are based on Xerces).

To do the parse we need a DOMParser object. We then call the parse() method, giving it a name of an XML file. The program below shows a minimal Xerces DOM program (Xerces has two sides to its personality, also supporting the Simple API for XML, SAX, but this article ignores this face of Xerces), all error checking and exception catching have been omitted for the usual space and clarity reasons (but see later for details):

```
#include <util/PlatformUtils.hpp>
#include <parsers/DOMParser.hpp>
#include <string>
#include <iostream>
int main() {
  // Initialise the XML processor
  XMLPlatformUtils::Initialize();
  std::cout<<"Enter the name of an XML file:";
  std::string filename;
  std::cin>>filename;
  DOMParser  parser;
  parser.parse(filename.c_str());
}
```

## Reading Elements and Text

The following code fragment reads all the <Entry> elements from the previous XML document:

```
DOM_Document phonelist=parser.getDocument();

// Get the <PhoneList> element
DOM_Element root
         = phonelist.getDocumentElement();

// Get all <Entry> elements into a DOM
// "nodelist" structure
DOM_NodeList entries
         = root.getElementsByTagName("Entry");

// Now iterate through the nodelist,
// processing each element found there
for (unsigned long node_index=0;
     node_index< entries.getLength();
     node_index++) {
  //Deal with entry
}
```

You will notice that the `getElementsByTagName` method provides us with a `NodeList` rather than an `ElementList`. This is because the DOM defines a list of nodes (which are all elements in this case) but not a list of elements as such. We have to deal with a base node object rather than a concrete Element object. More about that in a moment, but first let's see how we extract the name and number from those entries using the DOM `firstChild` and `nextSibling` attributes:

```
for (unsigned long node_index=0;
     node_index< entries.getLength();
     node_index++) {
  // Get each <Entry> element from the list
  DOM_Node node=entries.item(node_index);
  // Get the <name> node - the next sibling of
  // <Entry>'s first child node
  DOM_Node name_node
      = node.getFirstChild().getNextSibling();
  // Get the text node - a child of the
  // <name> node
  DOM_Node name_text
      = name_node.getFirstChild();
  // Finally get the actual text
  DOMString name=name_text.getNodeValue();

  // Repeat for <number>
  DOM_Node number_node
      = name_node.getNextSibling().
                          getNextSibling();
  DOM_Node number_text
      = number_node.getFirstChild();
  DOMString number=number_text.getNodeValue();

  // Process name and number, remembering they
  // are Unicode strings encoded in UTF-16,
  // whatever the XML declaration of this
  // particular document says
}
```

The above code shows that, although `firstChild` and `nextSibling` are attributes (not methods) of a DOM `node`, in this particular DOM parser (Xerces) they are accessed via member functions `getFirstChild()` and `getNextSibling()`. This is because the DOM specifies its interfaces as IDL (as used by CORBA [similar to that used by COM] ) and does not specify the technology to be used to implement these interfaces. The Xerces Parser chooses to define objects with `get/set` functions to represent attributes – like COM – hence the names used. This is implementation-defined though, and another parser may well access attributes of DOM objects as simple member variables.

As mentioned earlier the DOM `Node` interface is an alternative to the individual node-type interfaces. In as much as each type of node "is a" DOM Node, we have an inheritance relationship, easily implemented using a base `Node` class and derived `Document` class, `Element` class, and so on. However Xerces chooses not to implement it like that. For whatever reason (probably related to the fact that Xerces was implemented in Java before C++) you can only convert an Xerces `DOM_Node` type to a `DOM_Element` by using a `static_cast`:

```
DOM_Node some_node;
DOM_Element elem;
/*...assign some_node...*/
elem = dynamic_cast<DOM_Element&>(some_node);
                                    // WRONG!
if(some_node.getNodeType()
                  == DOM_Node::ELEMENT_NODE)
  elem = static_cast<DOM_Element&>(some_node);
                                    // OK!
```

This means you must check first whether the `DOM_Node` really does represent an element, and not a Text node for example. This must be done using the `DOM_Node::getNodeType` member function. If you try to "down-cast" a Xerces DOM node to the wrong kind of derived type beware – you won't get a bad_cast exception, and you will almost ceratinly get a crash if you try to call member functions on the wrong node type. Note that this is a detail of the Xerces implementation and other implementations may well use "real" C++ hierarchies in which this restriction does not apply.

You can also see in this case that within an <Entry> node, the <name> node is not the first child, but the sibling of the first child. Similarly the <number> node is not <name>'s next sibling, but the sibling of that sibling node. This is because the first child of <Entry> in our XML document is a text node holding the End-of-Line character(s), as is the first sibling node of <name>. Whitespace counts! Some parsers can be set to ignore these nodes but you need to be aware that they exist, and allow for them if necessary.

## The DOMString Interface

The document object model defines a string type to hold character data, which in Xerces is called `DOMString`, as you can see above. The `DOMString` class can *not* be directly output using normal `ostream` inserter operators since it holds Unicode characters encoded in UTF-16. As the comment at the end of the code extract above says, this applies whatever encoding your XML document uses, so you need to convert it to a suitable representation if you need to output it. In the case of Xerces a static member function called `transcode` is provided that returns you a pointer to an allocated buffer containing the string's local representation. This means you can implement an `ostream` inserter like this:

```
std::ostream& operator<<(std::ostream& target,
                         const DOMString& s) {
  char *p = s.transcode();
  target << p;
  delete [] p;
  return target;
}
```

This is not ideal, since you are responsible for deleting the memory allocated by the `transcode` function, and problems can occur if (for instance) it was allocated from a different heap – as may be the case in Windows debug environments, but it suffices for our illustration. See [5] for more details.

Other implementations will use different ways of transforming between different character encodings, which you will have to use since DOMString is mandated to use UTF-16 internally. Note the IBM version of Xerces, XML4C, has more Unicode support and may be worth looking into for those who need it.

You will get `DOM_String` objects from any methods that allow you to access text content. We have already seen the general `getNodeName()` and `getNodeValue()` methods of the `DOM_Node` interface, which are general calls whose return values depend on the node type, giving for example the text content of a text node, the name of an element,. There are also methods on specific node types that return text in the form of `DOM_String` objects:

```
DOM_Element::getTagName()
DOM_Element::getAttribute(name)
DOM_Attribute::getName()
DOM_Attribute::getValue()
DOM_CharacterData::getData()
```

And so on. The `DOM_CharacterData` class is actually a base class of two other types of node, the Comment node and the Text node. Actual DOM structures will never contain `CharacterData` nodes. You can get to the text data of both of the derived node types using the base class call to `getData`.

Note also that it is not guaranteed that a single block of text is contained in a single DOM Text object. When parsing, some implementations will

break text into a number of Text objects split at line breaks, entity references or other places.

## Attributes

There are two ways to get at attributes on XML Elements using the DOM. First, using the node interface, we can access the attribute objects attached to an element: Each <**Entry**> element in the example above has an attribute called **type**. In the DOM, an attribute is *not* a child node of the element it applies to (bizarrely though the parent of an attribute *is* that element). Suppose we are only interested in those entries whose **type** attribute has a value of "external". We can examine the type attribute on each iteration and only process those elements that have the required value. As already mentioned, the DOM does not insert attributes as child nodes of elements, rather it makes available for each element a list of its attributes. This list is of DOM type `NamedNodeMap` and works like a C++ `std::map` for accessing nodes by name:

```
// Find element with attribute 'type="internal"'
// in list of elements 'entries'
for(unsigned long node_index=0;
    node_index<entries.getLength();
    node_index++) {
  DOM_Node node=entries.item(node_index);
  DOM_NamedNodeMap attributes
                  = node.getAttributes();
  // Are there any attributes?
  if (attributes!=0) {
    DOM_Node attr
        = attributes.getNamedItem("type");
    // Is there a "type" attribute of the
    // right value?
    if ((attr!=0) && (attr.getNodeValue()
                      .equals("internal"))) {
      //Phew, got there!
      //Process this <Entry> node..
    }
  }
}
```

This method is useful if we don't know what attributes there may be, or if we want to get the attribute as a node object for some reason. There is a quicker way of accessing a named attribute directly:

```
DOM_String type_attribute
             = element.getAttribute("type");
if (type_attribute.equals("internal") {
  //process
}
```

## Creating XML using the DOM

We have seen how to parse XML using the DOM, now lets see how we can use the Docunent Object Model to build a representation of our own XML document.

You can only add XML to an existing document or document fragment, so first we need to create a new document. The Document Object Model does not actually define how implementations create an initial document object (which is of course a type derived from `DOM_Node`). In Xerces this is done using a method of an instance of the `DOM_DOM_Implementation class`:

```
DOM_DOMImplementation impl;
DOM_Document doc = impl.createDocument(
    0,              // Namespace URI if required
    "PhoneList",           // Root element name
    DOM_DocumentType());  // Default document
                          //   type object
```

The constructor of `DOM_Document` does not create you a valid XML document node, only a "shell" which must be filled in by assigning the result of the `createDocument()` function as shown above. The only thing you can do to this object is assign to it. Once this has been done, the document object will contain a single child, the root element node of the document.

## Adding Elements

To add elements to any exsisting element, including the root document element we need to use the `createElement()` and `createTextNode()` members of `DOM_Document` to create new nodes. Unlike creating a new document, once we have a document the DOM does define these methods as a way to create new nodes of varying types. Then we append the new nodes as children:

```
// Create an <Entry>
DOM_Element Entry
          = doc.createElement("Entry");
// Create a <name>
DOM_Element  Name
          = doc.createElement("name");
// Create a <number>
DOM_Element  Number =
doc.createElement("number");
// Add the <name> to the <Entry>
Entry.appendChild(Name);
// Append the <number>
Entry.appendChild(Number);
// Get the document element from the document
DOM_Element DocumentElement =
doc.getDocumentElement();
// Add the composite <Entry> element to it
DocumentElement.appendChild(Entry);
// Create a text node holding the string
// "Bond"
DOM_Text NameValue =
doc.createTextNode("Bond");
// Append the text to the <name> element
Name.appendChild(NameValue);
// Do the same for <number>
DOM_Text NumberValue =
doc.createTextNode("007");
Number.appendChild(NumberValue);
```

## Adding Attributes

To add the "type" attribute to the <Entry> element we could create an attribute object, but it is easier to just call the `setAttribute()` method on the element:

```
Entry.setAttribute("type", "secret");
```

This member of `DOM_Element` makes it easy to set attributes but at the expense of a little flexibility. The text passed to the function must be correctly encoded with no entity references. You can alternatively create an Attribute node in a similar way to how the Element nodes were created, add Text and any Entity Reference nodes to it, and call the `setAttributeNode()` method of the `Element` interface:

```
DOM_Attr Attr=doc.createAttribute("type");
Attr.setValue("secret");
// Create any Entity references in the
// attribute text and add them too...
// ...now add the attribute to the element
Entry.setAttributeNode(Attr);
```

If you want to create an attribute node, and the text doesn't contain any entity references, you can use the `setValue("text")` method of the `Attr` interface instead of adding text nodes to it.

## Outputting XML

Now we have built this tree in memory, we need to output it. How you do so depends on the parser. Xerces does not provide a standard ostream inserter for a `DOMString`, so we will have to provide one ourselves. This is most likely because, as discussed above, the character data in a `DOMString` object is stored in UTF-16 encoding, which is normally not what we output to plain text files.

This means that to output the contents of a DOM Node to an `ostream` we need do a couple of things:

1. Establish what kind of node it is (document, element, text and so on) and act accordingly
2. Convert text to the appropriate encoding used by the current locale.

The first can easily be done using a `switch` statement and involves outputting appropriate text for the different node types. Some nodes (e.g. document, element) contain other nodes so they would use recursive calls to the output function until the whole sub-tree had been dealt with:

```
ostream& operator<<(std::ostream& s,
                    DOM_Node& node) {
  switch (node.getNodeType()) {
    case DOM_Node::TEXT_NODE:
      s<< node.getNodeValue();
      break;
    case DOM_Node::ELEMENT_NODE:
      s<< '<' << node.getNodeName();
      /*..deal with attributes..*/
      s<< '>';
      /*..deal with child nodes
          recursively..*/
      s<< '</' << node.getNodeName() << '>';
      break;
    /*.. and so on ..*/
  }
  return s;
}
```

A full example can be seen in the Xerces sample program "DOMPrint.cpp".

This deals with the node type, but an additional `operator<<` is needed to actually output the contents of DOM String variables, and will be called by the function above. It needs to expand the special characters `&`, `<`, `>`, `'`, and `"` into predefined entities and send the contents of the modified string to the stream. I will leave that as an exercise!

## Validation

Any XML parser worth its salt should be able to validate the XML we pass it. There are two requirements we could have checked for us:

1. Is the XML well-formed XML (ie. does it conform to the W3C XML specification)?
2. Is the XML valid (does it conform to its DTD)?

The first requirement is the most basic – any XML we parse should be well-formed or we can't call it XML. The Xerces parser has two options and will either throw an exception or call a user-supplied error handler if it finds any irregularity in the XML it is parsing.

The error handler is an object of some class derived from `HandlerBase`. There are three severities of error, the parser will call the appropriately named method of your class, passing a reference to a `SAXException` object describing the error.

The second requirement depends on us having a DTD for the XML we are parsing. So far, the XML we have seen has been anonymous – that is with no document type specification. Let's look at the DTD for our example phone list:

```
<!ELEMENT PhoneList (Entry*)>
<!ELEMENT Entry (name,number)>
<!ATTLIST Entry type CDATA #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT number (#PCDATA)>
```

We will assume that this exists in a file called PhoneList.dtd. We can refer to this in out phonelist.xml file with a DOCTYPE declaration:

```
<!DOCTYPE PhoneList SYSTEM "PhoneList.dtd" >
```

If our parser is validating the XML (with Xerces this is enabled by calling the `DOM_Parser` member `setDoValidation()`, before parsing, with the value `true`) it will compare the XML we parse with the DTD given. Some parser-specific action will take place to report the errors, for instance with Xerces the error handler (previously specified, or a default) will be called, and the parse aborted.

Let's modify our parsing code accordingly:

```
//First an error handler,
//derive from Xerces HandlerBase
class MyErrorHandler : public HandlerBase {
public:
  void error(const SAXParseException &e) {
    std::cerr << "ERROR at line"
              << e.getLineNumber()<<std::endl;
  }
  void fatalError(const SAXParseException &e) {
    std::cerr << "FATAL ERROR at line "
              << e.getLineNumber()<<std::endl;
  }

  void warning(const SAXParseException &e) {
    std::cerr << "WARNING at line "
              << e.getLineNumber()
              << std::endl;
  }
};

...

//Now, tell the parser to validate the XML
parser.setDoValidation(true);
try {
  //Create an error handler
  MyErrorHandler error_handler;

  //And tell the parser to use it
  parser.setErrorHandler(&error_handler);

  //Now parse the file
  parser.parse(filename.c_str());
}
catch (const XMLException& e) {
  std::cerr << "An exception occured during
               parsing\n   Message: "
            << DOMString(e.getMessage())
            << std::endl;
}
```

The error handler method `error` will be called in the event that the XML does not conform to the DTD. The `fatalError` method will be called in the event of ill-formed XML, implying that the parse cannot continue.

The catch block will catch any internal errors that occur during the parse, or exceptions thrown from within your custom error handler.

## DOM Exceptions

You might have noticed that the exception objects passed to the error handler above were objects of type `SAXParseException`, not `DOM_Exception`. The Xerces parser uses objects of this class to encapsulate general parsing errors. I would guess that this could be because Xerces uses SAX internally when parsing an XML document into a DOM structure. The Document object model does have its own `Exception` class that is supposed to be thrown under various error conditions – the W3C recommendation states "when an operation is impossible to perform" but allows implementations to use "native error reporting mechanisms" if exceptions are not supported. It also says that general DOM methods return specific error values rather than throw exceptions.

Xerces will throw `DOM_DOMException` objects when you are manipulating DOM data structures or creating them from scratch. For example, you will get a `DOM_DOMException` if you attempt to substring a `DOMString` object with too high an offset.

This means that you should be prepared to catch a `DOM_Exception` object (but not during a parse, at least with Xerces) but it probably means a serious problem rather than a simple error like an ill-formed XML document.

## XML Namespaces

The DOM level 2 (the latest version) includes support for XML namespaces. In practice this means that, in addition to the standard DOM accessor functions we have seen there are some new ones that allow you to access namespace features.

For example:

```
// Retrieve the identifier of the namespace
// a node belongs to
DOMNode::GetNamespaceURI()
```

[continued at foot of next page]

# 4DML Revisited

**Silas Brown** <ssb22@cam.ac.uk>

I'm not happy with the way I explained 4DML in C Vu 14.4 p26, particularly the way the "depth'' dimension seemed to be grafted on to the design almost arbitrarily. My description followed the historical order of the design process, but it would perhaps be clearer if we ignored this and took another approach.

## Attributes as Co-ordinates

If you want to represent a sparse N-dimensional matrix of objects (i.e. an N-dimensional space that contains objects), then each object must somehow be associated with N numerical values, to position it on each of the N dimensions. You can think of these N values as extra attributes that the object has. You can also do the reverse: Take a collection of objects that all possess certain numerical attributes, and treat these attributes as dimensions in N-space; an object's co-ordinates are given by its values of the said attributes.

It is, of course, possible to do this with attributes that are not numerical, so long as they are sortable, or at least comparable so that two objects that are equal in some attribute can be put at the same co-ordinate on the axis (dimension) that corresponds with that attribute. However, 4DML only uses numerical values, for reasons I'll come back to later.

More generally, objects can have differing numbers of attributes; they are not limited to having a fixed number of them. If we imagine a non-Euclidean space in which objects can have positions on some dimensions but not others, then this space can represent a general collection of objects with arbitrary (numerical) attributes. Working with a non-Euclidean space does make the geometry slightly more interesting, but it turns out that a surprising number of operations are conceptually unchanged.

We now have a way of taking a near-arbitrary collection of objects, arranging them (in N-space) by their attributes, and doing geometric operations on this arrangement. This does not add any functionality, but a lot of people find geometry easier to think about than algebra, so expressing a problem in geometric terms can help.

## Trees as Attributes

The next thing to note is that an object's position in a tree can be represented by a list of attributes. For example, one attribute can indicate which one of the top-level branches the object is to be found under; another attribute can represent which one of the second-level branches to go down; and so on. Since these attributes can also correspond to positions in the non-Euclidean N-space, this shows that the N-space has all the functionality of a tree.

This is significant, because matrix-like data (such as a musical score) is often marshallised (that's "serialized'' in Java-speak) into a hierarchical (tree-like) representation such as XML, and then programmers try to work with it as a tree (by using re-writing systems and so forth) without being free to perform as many geometric operations as they could have done on the geometric representation. However, the above method, of representing the tree as attributes and attributes as geometry, provides a 'natural' way of converting between the two representations. This can make serialized matrix-like data considerably easier to process (and to transform into completely different serializations), and it can also manifest benefits when working with data that would not normally be thought of as matrix-like. As a bonus, it is possible to represent several independent hierarchies over the same data just by merging their attribute lists (so long as there are no naming ambiguities).

## Labels for Co-ordinates

Now, the problem is that having attributes such as "which top-level branch to go down'', "which second-level branch to go down'' and so on is not sufficient if those branches have names as well as numbers and we want to be able to set constraints on both. It is, of course, possible to store these names in extra attributes, or additional objects, or in some other manner, and all kinds of complex constraints could be designed into the system to make sure that they are used sensibly, but that might make the design too complicated. I preferred to support attributes with pairs of values, that is, each attribute can have both a named value and a numerical value. This can be regarded as attaching a label to each one of an object's co-ordinates in the N-space. One obvious way of extending this is to support more than one label on each co-ordinate, but people who need that functionality perhaps really should be thinking about using additional attributes instead.

4DML (four-dimensional markup language) is so-called because the prototype represents its non-Euclidean N-space as a set of four-dimensional points; each one of those points gives a reference to the object that it is helping to describe, the attribute that it is setting, the value, and the label. For historical reasons the prototype calls these "scope'', "depth'', "position'' and "name'' respectively, and lists them in reverse order. 4DML can represent trees and matrices, and blurs the distinction between the two; it can also be hacked to represent arbitrary relationships between objects (objects can be regarded as related if they share a common value of a certain attribute, i.e. they are at the same position on a certain dimension).

## Not a 'Real' Database

Earlier, I mentioned that 4DML only treats numerical values as co-ordinates, although it does support labels. It is important to realise that 4DML is somewhat different from a conventional database. In most databases, an object can have attributes of various types and the attributes store the data; for example, a record about a book might have as an attribute (or 'field') the author's name(s). It might then be possible to sort the database by the "author names'' attribute. If you wanted to do the equivalent in 4DML, you would have to give each author a position number (effectively pre-sorting them), and then arrange for the name and any other information, including all of the books that s/he wrote, to be stored at that position on an "author'' axis of the N-space. The point is that there is no information conveyed in the position (possibly excepting positioning information); it only serves to categorise and structure the information that is stored in the objects themselves (usually strings) which are opaque to 4DML's organisation. It's like using a database system in which everything is indexed by unique identifier. This keeps the design simpler and also helps prevent certain kinds of mistake (such as assuming that people's names are unique).

Although 4DML can be used for database-like applications, its main purpose is to represent documents in various notations. Documents are essentially collections of symbols with one or more reading orders (that is, sequences in which the symbols can be arranged); any markup over the symbols should reflect the arranging and interpretation of the symbols in the document, not the symbols themselves, and by extension, any values of any attributes associated with such markup should also reflect only this. It is possible to hack things differently, but I'd call that an improper use of the design. Perhaps I was wrong to use the analogy of objects and attributes to explain 4DML (it wasn't the way I designed it); it might make things easier to understand, but only if taken in the proper context.

*Silas S Brown*

```
// Retrieve the namespace prefix of a node
DOMNode::GetPrefix()

// Retrieve just the name part of the node
// name (omitting the namespace prefix)
DOMNode::GetLocalName()

// Retrieve a list of attributes with the
// given name, and the given namespace
// identifier ("*" means all namespaces).
DOMElement::GetAttributeNodeNS(
              const DOMString& namespaceURI,
              const DOMString& localname);
```

You can choose whether or not you want to use the namespace features of the DOM and stick to the appropriate set of method calls – those that support namespaces or those that do not.

*David Nash*

## References

[1] http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM
[2] http://www.w3.org/TR/REC-DOM-Level-1 and http://www.w3.org/TR/DOM-Level2-Core
[3] http://www.w3.org/TR/REC-DOM-Level-1/level-one-html.html
[4] http://xml.apache.org/xerces-c
[5] http://www.goingware.com/tips/xmlmemory.html

# Professionalism in Programming #16

## What's in a name?

**by Pete Goodliffe** <pete@cthree.org>

> "When I use a word," Humpty Dumpty said, in a rather scornful tone, "it means just what I choose it to mean – neither more nor less."
>
> Lewis Carroll [1]


Ancient civilisations knew that to name something was to have power over it. This was more than simply a claiming of possession. Some believed so strongly that they would never give out their own name to a stranger for fear they would be able to inflict harm using it.

Names mean an awful lot. It is fundamental to our concept of identity. We see examples throughout history. Even before 2000 BC we're shown Biblical examples of meaningful place names and children being given names to reflect circumstances. It's still convention for women to change surname when they marry, although the fact that some choose to do otherwise shows how they attribute significant meaning to their name. Why would people want to have their name changed by deed-poll if it meant nothing to them?

A name not only promotes identity, it also implies behaviour. Obviously a name doesn't entirely dictate what an object does. But it goes a long way towards defining how you interact with that thing, and how the outside world interprets it. This is borne out even more clearly by the fact we're never fixed to one name per 'object'. I'm known by different monikers in different contexts: the name my wife calls me[1], the name my daughter knows me by, the nick name I'm known as in chat-rooms, and so on. These names describe a relationship and interaction with me, and a role I fulfil.

A name marks something out as a distinct entity. It elevates it from an ethereal concept to a well-defined reality. Before someone put a name to electricity no one would have understood what it was, although they'd have some vague idea of its effects by watching lightning or Benjamin Franklin's demonstrations. Once named, it became identifiable as a distinct force and consequently easier to reason about. The Basque culture believes that naming something proves its existence. *Izena duen guzia omen da.* That which has a name exists [2].

Today the act of naming has become a multi-million pound business, used (with varying degrees of success) by small firms through to the largest multinational corporations to launch, re-brand and publicise products. The newer, ever more catchy names are intended to build awareness of their products and services.

So names clearly are of immense import.

As programmers, we wield this enormous power over the constructs we create when we name them. A badly named entity can be more than just inconvenient; it can be plain misleading and even downright dangerous. Consider as a very simplistic example the following C++ code:

```
void checkForContinue(bool shallWeContinue) {
  if (shallWeContinue) exit(0);
}
```

The parameter name is clearly a lie, or at least its sense is the other way round to what you'd expect. The function will not perform as anticipated and your program will halt as a consequence – a reasonably dire result from a single misnamed variable.

### Why should we name well?

Clearly we need to consider the names we give things carefully. The name creates a channel of understanding, control and mastery. Appropriate naming means that 'to know a name is to know the object'. And the opposite?

As in the real world, names can be both useful and limiting at the same time. People tend to stick to their initial perceptions of a concept, despite the proverb about judging books by covers. Therefore it's important to convey the right first impression through careful naming.

Apparently the human brain can only hold seven pieces of information concurrently (although I'm pretty sure I've got a couple of defective slots in my head reducing the overall capacity). It's hard enough to hold all the information about a program in your head as it is; we should not add complex naming schemes or require obscure references to make this task even harder.

### What do we name?

Let's spend some time thinking, as programmers, about what we name and how we name it. First: *what*? A minimum set of things we name, directly related to writing code, are:

- variables,
- functions,
- types (classes, enums, structs, typedefs),
- macros, and
- source files.

This list is by no means an exhaustive one – there are other higher level entities we'll put meaningful names to: states of a state machine, parts of messaging protocols, database elements, application executables, and so on. These five are enough to be starting with.

### What shall I call you?

So: *how* do we name? The naming convention for each of these classes of item should depend on the coding standard we're working to, if one exists. However, whilst a standard might mandate certain naming conventions it is not really sufficient to guide *appropriate* naming for each and every variable.

Generally there are very few rules from a compiler as to how we can name things. Modern languages have case sensitive names, don't allow 'white space' (spaces, tabs, newlines) and allow just alphanumeric and a few particular symbols (commonly at least an underscore). These days there are no appreciable limits on identifier length[2]. Without jumping through a considerable number of hoops we're usually limited to the ISO8859-1 (ASCII) character set, so non-English speakers are at a disadvantage. The C/C++ standards also reserve other ranges of names, for example any global identifier that begins with `str` or an underscore, and anything in a namespace called `std`. As practitioners it's important to be aware of these kinds of restriction so we can write robust and correct code.

Avoid jokey names like *blah* or *wibble*. They can easily creep in, and whilst amusing at first, just create confusion later on. Things like this are usually quick temporary hacks that outlive their expected uses. Name all things well first time, all the time. Obviously, being professional means that you don't explete when naming.

For each of the above listed sets of items, the following sections present some considerations for good naming.

### Naming variables

If a variable didn't consist of electricity it would be the sort of thing you could hold in your hand. It is very much the programming equivalent of a physical object, and a name that reflects this will usually be a noun. For example, some variable names in a GUI application might be `ok_button` and `main_window`.

If not a noun, it will usually be a 'nounised' verb, e.g. `count`. Numeric variables' names describe the interpretation of the value, e.g. `num_apples`. As we saw earlier, a boolean variable name might be a the name of a conditional statement, which is natural considering the value will either be true or false.

Since your variables are the fundamental data you work with you must give clear, very descriptive names. It doesn't matter if these are long if it's required to make their meanings unambiguous. '`a`' is not a realistic replacement for '`num_apples`'

However, there may be a case for short (even one letter) variable names: as loop counters. They actually make reasonable sense in small loops where variable names like '`loop_counter`' are not just obvious but can become rapidly tedious.

---

[1] This will alter depending on whether she's in a good or bad mood with me at the time!

[2] Be aware that older versions of C limited external unique linkage to the first six characters, and case was not *necessarily* significant. You need to understand exactly what the target of your code is when you write it.

When working with OO languages there are a number of conventions you may adopt to 'adorn' member variables to show they are members and not an ordinary local variable or (evil) global variable. This is a mild form of Hungarian Notation (see later section). Whilst not strictly necessary, some programmers find it a useful practice. In C++ some common forms are to prefix member variable names with an underscore, suffix them with an underscore or prefix with '`m_`'. The former method is frowned upon because it sails close to the wind; remember you can't have global identifiers beginning with underscore. Besides, a leading or trailing underscore makes the variable pretty unnatural to read.

Of course, this kind of member naming convention won't have any impact on a class' public API because all your member variables are private anyway (aren't they?).

The French language has two forms of the word 'you': *tu* and *vous*, depending on how familiar you are with a person. The name we know a variable by may depend on the context we need it in. For example, you may see a variable named differently in a function's public declaration (in a `.h` file) and in the implementation (in a `.c` file).

Some people feel it necessary to adorn pointer types with something like a '`_ptr`' suffix, and similarly for reference types '`_ref`'. This is another subtle incursion of Hungarian notation, and is redundant. The fact the variable is a pointer is implicit in it's type. If your function is so large that you think this adornment is a useful aid-memoire, then your function is probably too long!

Another commonly seen variable naming practice is using acronyms as a concise 'meaningful' name. For example you might declare a variable like this: `SomeTypeWithMeaningfulNaming stwmn(10);`

No matter what your method of variable naming, it is helpful to prefer a convention that distinguishes type names from variable names. Commonly type names have an upper case initial letter, and variables a lower case one. When using this convention it's not uncommon to see variable declared like this: `Window window;`

## Naming functions

If you hold a variable in your hand, the function is what you do with it – you don't just want to hold it forever. Since a function is clearly an action, its name will logically be, or will include, a verb to indicate this. A function name that was just a noun would not be clear: for example, what does '`apples()`' do? Does it return a number of apples, does it convert something into apples, or does it make apples out of thin air?

Meaningful function names will avoid including the words *be*, *do* or *perform*. These are a classic trap for students when first trying to consciously include verbs in their function names (*this function* does *XXX*…). That kind of word is just noise and don't add any value to the function name.

A function should always be named from the viewpoint of the user – hiding all the internal implementation stuff neatly away (that's the point of a function, it's a level of compression/abstraction). Who cares if behind the scenes it stores an element in a map, makes calls over a network, builds a new computer and installs a word processor on it, or whatever. If the user only sees the function count apples, the function should be called `countApples()`.

When we write functions they should be well documented (either in a specification or using some literate programming method). However, this is no excuse for not making the function name a clear statement of what the function does. Its name is part of its 'contract'. For example, what does `void a()` do? It could be anything.

The detail that must be included in the name will depend on the context it is defined in. For example, if a function that returns the number of apples in a tree is defined in a C++ class `Tree` then it needn't be called `numApplesInTree()`. It's full name would be an unambiguous description: `Tree::numApples()`. This context information works similarly for namespaces[3].

One final set of functions deserve consideration: "getters" and "setters". We see that some classes naturally act as collections of variables that behave like 'properties'. Each property needs a member function to read its value, and one to set it; some languages have built-in support for this kind of property. Whilst some argue that the existence of such get/set methods shows a weak design, nonetheless we see a lot of classes written containing this kind of API. There are a number of conventions related to naming these member functions: they include (for some property called *foo* of type *Foo*):

```
Foo &getFoo();
void setFoo(Foo &foo) const;
```

and:

```
Foo &foo();
void setFoo(Foo &foo) const;
```

or perhaps,

```
Foo &foo();
void foo(Foo &foo) const;
```

Your choice may vary, or again be dictated by your coding standard. This is one where, personally, I would violate the 'name always contains a verb rule' and go for the second option, since it reads the most naturally in code.

## Naming types

The sort of types we may create depends on the language we're using. In C we can only define typedefs, which are synonyms for other type names. You use them to provide an easier, more convenient name for existing type. It stands to reason, then, that a typedef should be clearly named. Even if it's only a local typedef in a function body it should still have a descriptive name.

Java, C++, and other OO languages are profoundly based on the creation of new types (classes). In the same way correct names for variables and (member) functions is vital to the readability of the code, good type names are paramount.

There aren't many obvious naming heuristics for classes, though. A class may be describing some state-full data object. In that case its name will probably be a noun. It may be a function object or class implementing some virtual callback interface. Here the name will probably be a verb, perhaps including the name of some recognised design pattern [3]. If the class is a bit of a mash of both, it's probably hard to name and potentially badly designed.

We saw a few words to avoid in function names, there are similar cases here. When putting a name to a class you should almost always avoid including the word *class* or *object*. In type names these are usually redundant noise. For example, `DataObject` is a bad name: the class may very well contain data, but it's obviously going to be used to create an object, that doesn't need restating in the type's name. The class name should describe the *class of data* and not the *actual object*. That's a subtle distinction, but important.

A bad class name can serve to really confuse programmers. As an illustration I've worked on an application which contained a state machine implementation. For some historical reason the base class of each state was called `Window`. It was very odd to work out what exactly was going on (and this wasn't helped by a distinct lack of documentation to boot). To add insult to injury the base class of a command pattern was called `Strategy` when it actually wasn't implementing a strategy pattern. Suffice to say it

### Capitalisation Conventions

Naming conventions are a source of about as many programmer fist-fights as the Eternal Holy Editor Wars (no one seems to have noticed that *vim* won years ago :-). *(Clearly a typo: you meant emacs, of course. — ed)* Most languages prohibit us from using white space and punctuation in our identifiers, so we adopt a convention for splitting up multiple words. There are a number of common ways of doing so which you'll see in modern code.

**camelCase**

As seen used extensively by the Java language libraries, also in many C++ codebases: KDE for example. It is so called because the capitalisation resembles a camel's humps, and was probably first used in Smalltalk in the early 1970s.

**ProperCase**

This is a close relative of camelCase, its only difference is that the first letter is also capitalised. Often the two conventions are used together. For example, in Java, class names are written in proper case, and variables and methods in camel case.

**using_underscores**

Proponents of this style are the implementers of the C++ standard library (look at all the names in the `std` namespace) and the GNU foundation.

There are, of course, more forms. How many can you come up with from the top of your head? You can start by mixing proper case with underscores. There are other similar naming considerations, like:

- How many vowels do you drop to make an identifier easy to type? Too many and it becomes unreadable.
- Do you require that any verb must come first in multi-word function names?
- Do you adorn member variable names, and if so, how?

[3] In fact, this is a reasonably universal principle that could apply to most named items. For example enumeration element names found at class scope would be different to a similar definition at global scope.

took me a little while to get my head around what was going on. Better naming would have allowed me easier access to the code's logic.

## Naming macros

Macros are the walnut-cracking sledgehammers of the C/C++ world. They are a basic text search/replace tool that don't respect scope or visibility. They're tactless. However, there are some walnuts that just won't crack without them.

Since they have such drastic effects there is a well-established tradition for naming macros in a maximally obvious way, using CAPITAL LETTERS. Follow this without fail. And don't make any other name entirely capitalised. This makes macros stand out like a sore thumb, which is basically what they are.

## Naming files

Did you think of files when we talked about naming things? The name of your source files can have a real effect on the ease of coding. Obviously what you call a source file, be it a header or implementation file depends on what goes in it. In C and C++ there aren't actually any restrictions on file names, but calling headers "`something.h`" is such a universal convention that it would be like sticking pins in your eyes not to. We already feel some pain from the lack of rigid definition though. Different people call C++ implementation files different things, `.C`, `.cc`, `.cpp`, `.cxx`, and `.c++` are common file suffixes. Your choice will usually depend your compiler, personal preferences, and/or coding standard. I have even worked on platforms that didn't support file extensions and defined file types by the name of the enclosing directory (with appropriate massaging for standard header file includes). That was reasonably evil!

Moving past the discussion of what suffix to give your files, exactly how should you name them? To make this naming easy and obvious a file should usually contain one conceptual unit. Any more stuff in that one file is asking for trouble in the long run. Split your code into the maximum number of files you can, not only will it make them easier to name, it should reduce coupling since you don't `#include` one big monolithic header file who's smallest change in one dusty corner requires many dependant recompilations. If you have a file defining the interface for a *widget* it should be called "`widget.h`" (not "`widget_interface.h`", "`widget_decls.h`", or any other variation).

Once you have a file that can be appropriately named, you conventionally should balance each `foo.h` with a matching `foo.cpp` that implements whatever the `foo.h` declares. This is both obvious and conventional.

Now, there are other insidious issues when naming files. You need to sort out the capitalisation. Some filing systems (naming no names[4]) can't get this right, ignoring case when looking up file names. When porting code to platforms where case is important your code won't compile unless you've observed capitalisation carefully. Perhaps the easiest method of avoiding this sort of issue is to mandate all lowercase filenames. If you don't, be careful. For the same reason, if your filing system considers that "`foo.h`" and "`Foo.h`" are different files, don't exploit it. Make sure that your filenames differ by more than just case. If you mix languages in a single project don't create `foo.c` and `foo.cpp` – it's messy; which file is used to create `foo.o`?

Older filing systems limited the number of characters you could use in a filename, which made naming much messier. Unless you have to port code across to such an archaic system this kind of limitation can be safely ignored.

Try to ensure that each header file you create has a distinct name, even if they're all spread across different directories. This makes it easier to reason about which header file you are actually including when you `#include "foo.h"`. If there were two different files with the same name a newcomer to the codebase would be confused. This gets to be more of an issue the larger the codebase gets. One valid way to work with this is to include some path information in the logical filename, i.e. you may include "`library_one/version.h`" and "`library_two/version.h`" without too much panic.

As an illustration of how file naming impacts ease of coding, I worked a particular project where the majority of the filenames matched the class names exactly, for example the class `Daffodil` was defined in `daffodil.h` (names have been changed to protect the guilty). To make things more interesting, every now and again a file was named in a *slightly* different manner, usually slightly abbreviated, so `ProxyObject` would be held in `proxyobj.h`. That just made finding the right filename to include more complicated and time consuming than it needed to be. On top of this, not all of the `Daffodil` class implementation was necessarily in `Daffodil.cpp` – some of it might have been in a shared `FlowerStuff.cpp` and perhaps also in `Yoghurt.cpp` for no adequately explained reason. As you can imagine, this made finding particular bits of code a nightmare.

[4] Painfully obvious pun.

## A rose by any other name

That's a pretty large set of considerations for naming bits of code. What are the overall principles to pull out? Perhaps the most important thing is that you should ensure consistency in all of your naming, and not just within your own work, but also respecting company-wide principles. This goes right down to the typography of a name, and its capitalisation. For example I have no confidence in the quality of a class interface if it looks like this:

```
class foo : public Bar {
  public:
    doTheFirstThing();
    DoTheSecondThing();
    do_the_third_thing();
};
```

When you get a lot of people working 'together' on the same lump of code its very easy to end up in this state, being about as internally consistent as a random number generator. It's often a symptom of worse problems – the programmers probably aren't respecting the fundamental design of the code they're simultaneously working on. This is where mandated coding standards and central design documents are a big advantage.

With consistent naming we get code that is intuitive, therefore easier to work with, easier to extend and maintain. In the long run it's much cheaper to manage. Whilst the C++ standard library is a definitive source of programming best practice it also contains some classic examples of inconsistent and inappropriate naming. This shows that no matter how good your codebase you'll probably have to live with some bad naming.

There is power in a name, and power that allows us to be more expressive than a language's syntax alone might allow. Think about how you can use similar names to group things together, or how you can imply which of a function's parameters are input or output.

## Hungarian notation

Bunfight! There is nothing like the mention of Hungarian Notation in the programmer's realm of naming that will raise hackles and cause such a heated discussion. A few readers won't know what this practice is. Since it's such a controversial issue, in describing it I'll be careful not to make any judgement calls; it's not really the place of such an article.

Hungarian notation is the downright evil, obstructive and complex practice of encoding information about a variable or a function's type in its name in the misguided belief that it will make the code more readable and more maintainable. It sprang from Microsoft in the 80s and it's particularly interesting to note that these days large parts of Microsoft itself ignores this abominable convention. It's widely used in their public Win32 APIs and the MFC library, which is almost certainly the main reason for its popularity.

It's so called because it was pioneered by a Hungarian programmer Charles Simony. It's also called that because variable names written using it look like they may as well have been written in Hungarian: non-Windows programmers will be confused by surreal names like `lpszFile`, `rdParam` and `hwndItem` dotted around every piece of code.

There are many subtly different and not-quite compatible dialects of Hungarian Notation which doesn't help matters. Also in some situations, the same prefix can mean different things. These are some common Hungarian encoding prefixes, not including any magic Microsoft typedef codes:

| | |
|---|---|
| `p` | pointer to... (`lp` means 'long' pointer, an old architectural issue – if you don't know, don't ask) |
| `r` | reference of... |
| `k` | constant... |
| `rg` | array of... |
| `b` | boolean (`bool` or some C typedef) |
| `c` | `char` |
| `si` | `short int` |
| `i` | `int` |
| `li` | `long int` |
| `d` | `Double` |
| `ld` | `long double` |
| `sz` | zero terminated `char` string (note: *not* `p`) |
| `S` | `struct` |
| `C` | `class` (you can define your own class abbreviations too) |

# Linux Server Series Part 1

**Paul Grenyer** <pjgrenyer@iee.org>

## Preferred Linux Distributions

This is the first part of my series on setting up a Linux Server. As the series progresses I will explain how to set-up a Linux based server starting from choosing a Linux distribution through to choosing hardware and setting up, or upgrading to that latest version of, the likes of:

- Samba,
- Apache web server,
- A DNS server,
- A DHCP server,
- Internet connection sharing,
- A firewall,
- Postfix,
- GCC (Compiler),
- MySQL,
- CVS

for a small home sized network.

This first part of the series deals with choosing a distribution. The next part will cover choosing hardware and the third installing and setting up the Linux distribution. After that I will look at configuring the packages listed above, starting with Samba.

It doesn't make a great of deal difference to me which distribution of Linux I use for the server and there are many to choose from so I posted a query to accu-general to try to get an idea of which distribution people are using and what they would like to see the article based around Mandrake, SuSE, Redhat, Debian and Slackware came up again and again. Below I have summarized the general feeling about each of these distributions based on the comments I received.

### Mandrake

`http://www.mandrakesoft.com/`

The main points made about Mandrake were that it was very easy to install and very good at detecting lots of different pieces of hardware including some of the relatively more 'exotic' devices such as USB printers, DVD Drives and CD Rewriters. This makes it very good for Linux beginners.

### Redhat

`http://www.redhat.com/`

RedHat is a main-player in the USA, however it tends towards the non-standard edge of things. Not as easy to install as Mandrake, but seems to be in wider use, especially in industry. It has some other strange features such as installing postfix instead of sendmail as the MTA. A few problems were pointed out such as SCSI compatibility issues and faulty compilers.

Just about everyone mentioned RedHat at some point. Most didn't give a reason, but said that RedHat would be the first distribution they tried.

### SuSE

`http://www.suse.co.uk/`

SuSE is big in Europe, especially Germany. It equals Mandrake in terms of ease of use and installation and has a better hardware detection system than Mandrake. SuSE actively support XFree 86 development and KDE development (`http://www.kde.org/` - a very popular desktop).

Unlike RedHat, SuSE has printed manuals that actually have useful information, they also offer 60 days free support for their personal edition and 90 days for their Professional edition. SuSE has a much broader selection of ancillary packages and uses the concept of maintaining the configuration files using `rc.config`.

Again a lot of people mentioned SuSE, generally as the second distribution they would try after RedHat. All comments were be very positive and this distribution is obviously very popular.

### Debian

`http://www.debian.org/`

Debian has many fans in the USA, apparently due to the apt-get package manager that keeps the system well patched. However, it's strictly non-commercial and all the developers and maintainers are unpaid, so the release cycle can be lengthy.

Debian has always at least three releases in active maintenance, Potato ('stable'), Woody ('testing') and Sid ('stable'). Woody went into 'almost-frozen' a couple of months ago preparing for the new 3.0 release and will later become the new 'stable'-release.

### Slackware

`http://www.slackware.org/`

Slackware was mentioned by just a few people and is popular because it does not try to do things for you, like most of the other distributions. It has a very simple package manager that just requires tar and gzip. There is greater control over what is installed and Slackware is the only distrobution that is compiled for 386/486 "out-of-the-box".

However, Slackware has its strong proponents and some found it too sparse for their needs.

When I first read all the replies I received to my original post, it was clear to me that RedHat was popular so I decided I would use it on my server and have now gone ahead and purchased the distribution. However, now that I have looked more closely at what people have actually said, most people just mention RedHat as the distribution they would try first. The most 'popular' and highly thought of distribution appears to be SuSE, with no one having anything bad to say about the later versions.

I have some previous experience with RedHat and I am keen to give it a go. However, I am also keen to try SuSE as a result of the response to my post. Therefore, if people are interested I am prepared to do two versions of my Linux Server Series, one with RedHat and one with SuSE, they will more than likely overlap in many places anyway. If you would like to see this, please let me know (`pjgrenyer@iee.org`).

All of these distributions are available from the Linux Emporium (`http://www.linuxemporium.co.uk`) in the UK.

### Thanks:

Thank you to everyone who replied to my original post on accu-general. I hope I've got you all here: Neeraj Korde, Tim Pushman, Ewan Milne, Anthony Williams, Andy Leighton, Graham Whaley, Richard Moseley, Charles Polisher, Jason Gruber, Phil Hibbs, Kevlin Henney, /dev/null

*Paul Grenyer*

---

Hungarian notation was relatively unbearable in C (not to mention unnecessary once the language became more strongly typed), and can become rapidly nauseating in C++ since it doesn't really scale up to the many new type definitions you can introduce. If you really want to confuse a maintenance programmer use Hungarian notation and then go around a few months later changing the types of all the variables without search-and-replacing every single variable name (naturally, it will take too long to do that). Aside from being a joke, this is not an uncommon problem with this naming scheme.

Unless you are forced to use it, Hungarian notation is best left well alone. Naturally, thousands of readers will now write in and argue against such a neutral and diplomatic viewpoint (please do!).

## Conclusion

Our ancient ancestors knew it and good programmers know it. It's crucial to name things well. Good names serve more than just an aesthetic purpose, they convey information about the structure of code. They are an essential tool to aid comprehensibility and maintainability. Bad names have the potential to mislead. There *is* power in a name and an experienced professional programmer understands the balance of concerns involved when naming any part of their code.

This all comes back to the main reason we write code in high level-languages: to communicate. Our communication is to an audience of code-readers, that is other programmers, rather than to the compiler.

*I GOT PEELED OFF*[5]

## References

[1] Lewis Carroll (1832-1898). *Through the Looking Glass.*

[2] Mark Kurlansky. *The Basque History of the World.* Jonathan Cope. ISBN: 0-224-06055-4.

[3] Gamma, et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley. ISBN: 0-201-63361-2.

[5] Inappropriate naming courtesy of the Internet Anagram Server.

# Reviews

## Bookcase

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

### Francis Glassborow writes:

Please note the names of the reviewers in this issue's Bookcase. Notice anything? Well if you remove the reviews by me and by Chris Hills there is little left. Another thing that you may notice is that there are no books on straight C or C++. Personally I think that is a good sign as long as it is not always that way. The glut of ill-considered books aimed at novices (or whose authors fondly believe will be all things to all people) has moved away from C (quite a long time back) and from C++. The flood of books on Java is beginning to subside while everyone jumps on writing books about XML and C#. It would be a serious mistake to judge the popularity of a language by the number of books being written for it. What is more significant is the number of genuinely advanced books being written for a language.

Now let me reflect on my first question. ACCU needs more reviewers so that the current ones are not over-worked. We need those of you with specialist skills that can review some of the more obscure books publishers send to us. I do not know if you have realised but as long as the book is in first class condition, you can sell on any books that you do not want to keep (as long as you have first reviewed them). For example Blackwell's Bookshops would give you somewhere between 30% and 40% of the list price. A direct sale to a local retailer is probably better than selling second-hand through someone such as Amazon because the postal costs would eat up much of what you would receive if you went that route.

If you are not already reviewing books for ACCU, give serious consideration to doing so. just a couple of books a year would make a tremendous difference to the state of my office (I have over £5000 worth of books waiting for reviewers).

You know that I try to provide comparative costs for titles sold both sides of the Atlantic. This time I came across some quite bizarre cases. A couple of books were only listed in the hardback versions in the US even though paperback was available in the UK. In one case though, identical copies were much, much more expensive in the US. Where I give a comparative cost it is, as far as I can tell, for the same product. There is also the case that Pearson Educational has just substantially reduced some titles. And some titles are available from Amazon.co.uk at 50% discount.

*Francis Glassborow*
<francis.glassborow@ntlworld.com>

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
**The PC Bookshop (020 7831 0022)**
orders@pcbooks.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 some books shows the dollar pound conversion rate where known. I consider a rate of 1.5 or better as appropriate (in a context where the true rate hovers around 1.6). I consider any rate below 1.3 as being sufficiently poor to merit complaint to the publisher.

## Java & C#


**C# Bible by Jeff Ferguson et al. (0 7645 4834 4), Wiley, 798pp @ £29-95 (1.34)**
**reviewed by Francis Glassborow**

Several publishers have well defined series with a clear brand image. The 'Bible' titles seem to get passed around. IDG, The Waites Group and Sams have all published books claiming to be *<something> Bible*. Picking up this book with a cover dominated by yellow and red, and noting that it was a multi-author work made me think that it would be a Sams publication. It isn't, this one is from Wiley. Then the penny dropped, Wiley recently acquired the Dummies series (and I am waiting to see what they will do with such gripping titles as *Sex for Dummies*) This book is a very fat volume for dummies and post dummies. Try the first paragraph from 'Who Should Read This Book'

> This book was written with both the novice and experienced developer in mind. If you know nothing at all about the basics of software development, this book will get you started with the fundamentals, teaching you how variables, control loops, and classes work. The book will also speak to developers of any skill level, showing you the .NET tools available for C# development and providing you with tips to make your own C# applications work seamlessly within the .NET Framework development guidelines.

Does anyone actually believe that? If a book is suitable for someone who knows nothing about the basics of software development it will have nothing to offer the experienced developer. Even books written to introduce experienced developers to a new language are unlikely to be of use for anything else. I wish publishers would stop encouraging authors to target books at ridiculously wide readerships. The result is always that they fall between the needs of everyone.

Actually the first half of this book is clearly intended for the newcomer to programming. Unfortunately there are many other things apart from syntax that such a person needs to learn. Anyone who has done some programming will not be too uncomfortable with the jargon of programming (variables, functions, operators etc.) but the newcomer certainly does not want to be faced with terms like 'metadata' before they have even written a line of code.

So my conclusion is that the first part of this book is not suitable to the genuine novice. But it isn't suitable for the experienced programmer because in that case the pace is too slow and too much time is spent in inadequate (for the novice) explanations. The experienced programmer does not need to be told what a `break` statement does. S/he does want to know the syntax of it in this language together with where it can be used.

This problem manifests itself even when we move on to the more advanced sections. These just do not read as expert writing for expert. The text drifts, the authors waffle. The book is peppered with source code that is of poor quality, laid out with oodles of whitespace (and yet still lines wrap because neither the editor nor the author has thought about the consequences of nesting depth. Let me give you an example. The squares of a chessboard can be represented by a letter (a-h) followed by a digit (1-8). For internal purposes we would normally represent that by an ordered pair of integers each in the range of 0-7. How should you convert from the external representation to the internal one? Well I will lay you pretty good odds that most readers of this would not use a `switch` statement to convert `chars` to `int`s. I can think of several good ways that an experienced programmer would do it, but not with a `switch`. And should we be setting such examples to novices?

There is a very narrow readership for this book, those who are inexperienced programmers in at least one other language who do not want to learn a good coding style.

Sorry, we have enough poor books on programming without adding more to them just because we have a new language to learn about. I have no doubt that this book will be a reasonable commercial success and that many readers will be quite satisfied but that will be more a measure of the poor standards we have come to accept in IT publishing rather than due to any inherent quality of this book.


**Developing Applications with Java & UML by Paul Reed (0 201 70252 5), Addison-Wesley, 463pp @ £34-99 (1.29)**
**reviewed by Silvia de Beer**

Before I started reading this book I thought it would just be another introduction to UML. I was mistaken; it is actually a good case study on how to follow the Unified Process. The theory of the Unified Process and UML is very well interwoven with a case study. Attention is given to all development process concepts and they are put into a practical context. Only the most useful details of UML are explained.

A strong point of the book is that the design of use cases is well explained. This is

especially important for people new to the Unified Process, because this is the first point where developers can fail. They do not have a good idea what to achieve exactly. The book warns on the pitfall of too many small use cases, which, as the author explains, should only be a pathway through a use case. Some additional tools like event tables are described, to ease the transition to other phases in the development process.

General task lists and templates are given, after which the case study fills them, which serves as a concrete example. The case study handles an ordering system for musical instruments, which goes through the inception and elaboration phases. The various activities during those phases are shown, including a little coding during the early iterations. The transitions between iterations are well explained. It is a pity that no chapter has been dedicated to the construction phase, especially after the good efforts of going through all earlier iterations. Even if those phases were not worked out in detail, some text could have been dedicated to the implementation and transition phase.

Java is used to implement the case study. Java Beans, Servlets, JSP and Enterprise Java Beans are used to show different possible implementations during the elaboration phase. This might make the book slightly less useful if you have not worked at all with this technology and even more useful if you work in that area.

This book would be very well suited to get a small team of developers who start using the Unified Process, on one line. It could also be used to support a university project of a few students. A starting developer would get a very practical view of the essential tasks in the Unified Process.


**Developing Enterprise Java Applications with J2EE & UML by Khawar Zaman Ahmed & Cary Umrysh (0 201 73829 5), Addison-Wesley, 330pp @ £30-99 (1.29)**
**reviewed by Silvia de Beer**

I was disappointed with this book. It starts with an introduction to UML, after which an overview of the J2EE technologies is given; Servlets, Java Server Pages, Session Beans, Entity Beans, and Message Driven Beans. Based on the title I expected a book for experienced Java developers and advice on how to develop sound applications using UML as the modelling language. However this book is just another introduction to UML. One should read the title as that J2EE concepts are documented with UML diagrams.

To give an example, the Servlet life cycle is explained in a sequence diagram, showing init(), service() and destroy() messages to the Servlet object. The same is done for the various types of Enterprise Beans, their lifecycles and basic interaction are documented with UML diagrams. The example diagrams and implementations do not bear enough coherence. They seem to me like a first iteration in a design project, which is not correctly reviewed yet. Only the last 25 pages are dedicated to a small case study, which would not be enough to help the developer apply the Unified Process correctly.

One interesting point of the book is that the advantages and disadvantages of the use of various types of Enterprise Java Beans are discussed and how EJBs could interact with JSPs, Servlets and normal Java Beans. One should pay attention to the performance of EJBs though.

Concluding, if you would like an introduction to UML and J2EE technologies, this is a reasonable book. If you are not looking for that, leave this book aside. The book does not contain much original work. I did not find any tips on how to avoid pitfalls in the development process. The authors did not manage to inspire me, although they claim to have over ten years of software development experience. If I compare this book to Developing Applications with JAVA and UML by Paul Reed, I would definitely choose the latter.

**Java Precisely by Peter Sestoft (0 262 69276 7), MIT, 118pp @ £10-50 (1.42)**
**reviewed by Francis Glassborow**

[see web]

**Borland JBuilder 3 Unleashed by Neal Ford et al. (0 672 31548 3), SAMS, 1072pp+CD @ £36-50 (1.37)**
**reviewed by Steve Dicks**

A 1,000 page book on an IDE? A very unlikely scenario and so it proves in this book. This weighty tome tries to deliver 'Java everything plus JBuilder' and inevitably fails.

The first 150 pages try to cover software development in general and manage little more than this example on 'Best Practices':

> 'Development teams should make efforts to establish their own coding standards. These standards can consist of high-level practices such as the use of design patterns or low-level standards including the specific formats for program structures such as if blocks'

I.e. well intentioned but ultimately too shallow to be of any real use.

I also take exception to some of the author's 'example' code viz this little fragment on 'handling I/O Exceptions' (middle of the try block removed for brevity):

```
try{
// <do some file i/o>
}
catch (IOException iofex){
  Message msg = new
     Message(new Frame(),
           "File Exception",
           ("An exception
            occurred writing
            to the file!"));
  msg.setVisible(true);
  return ("Error");
}
```

This has so many faults [user told 'exception' rather than user language, no mention of filename or any detail from exception, no logging of exception, and the mysterious return of a string "Error" from the method (whose signature we never see) for the error case] that I would hesitate to call it an 'example'.

The book has 6 different authors, who do make a reasonable job of writing in a consistent style; if only the book as a whole didn't have such a 'global' ambition.

While there are some good chapters on use of JBuilder, they are so thinly scattered to make this book neither a good reference nor an introductory text. The publication date of 1999 just adds to this book's problems, that it is written at a snapshot in time which ages far too quickly in the Java world.

## Perl, Python etc.


**Making Use of Python by Rashi Gupta (0 471 21975 4), Wiley, 390pp @ £25-95 (1.35)**
**reviewed by Francis Glassborow**

What does the title signify to you? Now turn to the back cover where we find the following claim highlighted:

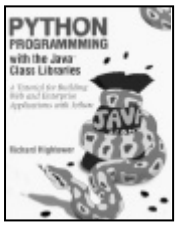
> A step-by-step guide on how to use Python for CGI scripting, GUI development, network programming, and much more!

[The exclamation mark is theirs, not mine]. Now when you come to check between the covers you find the first half of the book is the usual mix of preaching the virtues of Python (if I were not already convinced of that, why would I be spending time studying Python) with an introductory course on Python. Unfortunately I find that material very poorly presented and at times deeply confusing (please note that I already know some Python though I do not program fluently in it.)

The first half of this book clearly does not stand up to the promises of the title and the back cover. It would be much better for the author to assume that the reader has a basic knowledge of Python (or can get one from such excellent texts as 'Learning Python'). Now having made that assumption he can focus on what was promised. In other words the book should start at page 213. Now if you start reading there you will be faced with large quantities of HTML. Now think about this, what I really need is some instruction on HTML. OK, you (yes, you reading this review) already know that writing cgi scripts presupposes being comfortable with HTML but if that is the case why does half the chapter consist of reams of HTML? I want to know how to write a cgi script in Python and have just a single pathetic example buried away in a chapter littered with trivia.

The next chapter on Database Programming (using mySQL) is a little better but is still too broadly based and inadequately focused on the Python. If the reader does not already know about mysql they will not be able to make sense of this chapter, and if they do they will not be interested in most of the contents.

This is another example of a book where the author is so concerned about having a broad readership that he assumes that his reader needs to be spoon-fed. The author needs to have a clear image of whom he is writing for and discard all the superfluous floss that litters his book. He then needs to provide about ten times as much information directed at the real needs of the reader.



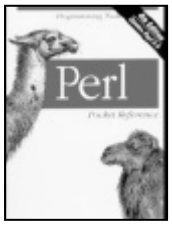

**Python Programming with the Java Class Libraries by Richard Hightower (0 201 61616 5), Addison-Wesley*, 620pp @ £37-99 (1.32)**
**reviewed by Francis Glassborow**

The sub-title is *A Tutorial for Building Web and Enterprise Applications with Jython*. That is a pretty good summary of this book as long as the reader understands that it is about a particular version of Python designed to run on a JVM. This version is often called Jython.

The book is a well-presented text for newcomers. The competent C++ programmer can skip the first few chapters that are concerned with really basic stuff. By the time we reach chapter eight we are ready to start exploring some of the power that lies under the Python hood in general and the Jython one in particular.

The point about Jython is that it compiles down to Java bytecode (or, as is common for simple Python, it can be directly interpreted as bytecode, which is an interesting idea because we finish up with an interpreter sitting on top of an interpreter, Python interpreted into bytecode which is then interpreted by the JVM). The great advantage of this is that the Java Libraries are also compiled to bytecode. The JVM does not care where bytecode comes from; it simply interprets it (or if you have a JIT, it compiles it on the fly). This means that as long as we can tell Python what a Java API provides we can mix Python with Java libraries. This is the reason for the title of this book.

If you have some programming experience (if not, I think something such as 'Learning Python' would be a better start) and want to explore the potential for using Jython for developing Web and Enterprise applications this book would be a good place to start. There is much more to Python than is covered in this book so do not assume that it will turn you into a Python expert. However if you put in the hours of study this book will help you realise the potential of Jython for simple to intermediate applications.




**Perl Pocket Reference 4ed by Johan Vromans (0 596 00374 9), O'Reilly, 91pp @ £8-95 (1.45)**
**reviewed by Francis Glassborow**

This is another of O'Reilly's small format books that really will fit in a pocket. I do wish O'Reilly would work out how to make books in this format lie flat, because as they are you have to hold them open with one hand while typing with the other.

If you need something to remind you of Perl syntax/semantics while away from your normal work area, this will do nicely. The first paragraph on page 1 says all you need to know when it comes to deciding if this is the version for you:

> The *Perl Pocket Reference* is a quick reference guide to Larry Wall's Perl programming language. It contains a concise description of all statements, functions, and variables, and lots of other useful information. This edition is based on Perl version 5.8.

**Beginning Perl for Bioinformatics by James Tisdall (0 596 00080 4), O'Reilly, 368pp @ £28-50 (1.40)**
**reviewed by Robert W. Hand**

This book is a curious, hybrid that teaches the computer language Perl, on one hand and serves as an introduction to the rapidly growing field of Bioinformatics, on the other. The author recommends the book as a 'practical introduction to programming for biologists'. Since our organisation is for C and C++ programmers rather than for biologists, this book and my review might seem 'off-topic'. However, I found the book to be excellent and the author pulls off the marriage of his two aims quite well.

The first three chapters contain very basic information for the novice programmer – how to get and install Perl, how to get a text editor, how to back up work. The advice is sound, but an experienced programmer could skim them without loss of direction.

At this point, I started to read Appendix B, Perl Summary. This appendix reads very well, provides a good reference to the language and is worth the price of the book. In particular, the pace of the presentation is consistent with the reading appetite of an experienced programmer. The section on regular expressions is especially noteworthy.

Chapters four through nine contain the meat of the book. The focus is on writing programs to analyse DNA, genes and proteins. As a result, some Perl topics are omitted and others are emphasised. For example, the important use of Perl in cgi programming is mentioned only in passing. The example programs do illustrate important programming principles while allowing illustrations of the power of Perl in the field of Bioinformatics. The approach is practical rather than theoretical. The requisite biology is explained in very simple and abstracted terms that should be understandable by the non-biologist.

Chapters ten through twelve introduce the GenBank, Protein Data Bank, and BLAST with good examples of Perl programs that manipulate such data. Unfortunately, the biological detail started to strain my knowledge of molecular biology.

This book was my first encounter with Perl. I found that I could read it quickly and there appear to be few errors. I recommend it to readers who are interested in the application of Perl to Bioinformatics.

## ADO, COM etc.


**Microsoft .NET for Programmers by Fergal Grimes (1 930110 19 7), Manning, 356pp @ £31-50 (1.11)**
**reviewed by James Gordon**

A neat little book, not too big and unwieldy. It assumes nothing and leads you from an overview of the .NET framework and compiling your first C# and VB.NET program. It describes the built in types and assemblies among other things of interest to understand before getting into the bulk of the book, which is a poker game.

The author uses this example to show how to produce a standalone application as well as a Web Service. It is all run from Visual Studio.NET and includes creating the projects and using the Visual Designer and debugger.

This book is not an in-depth look at .NET but shows enough to get you started and a bit more. I like the layout and not being padded out with great swathes of code it is easy to read and understand.

At over £30 it is expensive, but no more than many other books.


**ADO ActiveX Data Objects by Jason Roff (1 56592 415 0), O'Reilly, 601pp @ £31-95 (1.41)**
**reviewed by Paul Grenyer**

From the beginning I liked this book. However, once I'd finished reading it I still felt that overall it is a good book, but it is let down by its poor C++ examples (I feel unforgivable in a post C++ Standard publication) and its poor .Net chapter which is just plain wrong in places.

The contents shows that the book is divided into the usual few introductory chapters, one which explains how to use ADO from various different languages, and then goes on to look at some of the familiar ADO objects such as `Connection`, `Recordset` and `Field`. Towards the end of the book there is also a chapter on `ADO.Net`. The final third of the book is comprised of an ADO API reference and three appendices, Introduction to SQL, The Properties Collection and ADO Errors.

This book is aimed squarely at VISUAL BASIC developers. Although ADO: Active Data Objects describes a COM based technology and includes a chapter on creating and using ADO objects form other languages (such as C++), the remainder of the examples in the book are in Visual Basic. Many of the examples are difficult to translate directly into other languages, especially if you are not familiar with COM in that particular language. This is particularly true of Chapter 8: The ADO Event Modal, as COM events are handled for the user by the Visual Basic IDE.

If you are a Visual Basic developer or a C++ developer with previous experience of COM and the basics of the ADO C++ API, then I would recommend this book if you wanted to know more of the details of ADO and how to use them effectively.

## Embedded Programming

**Embedded C by Michael Pont (0 201 79523 X), Addison-Wesley, 294pp+CD @£24.99 (1.80)**
**reviewed by James Dennett**

When opportunity arose to provide a second review of Michael J Pont's book "Embedded C" I welcomed the chance, given that I have been working in the embedded world since the beginning of 2002 and still have plenty left to learn. This review have been written based on my reading of the book only, and do not take into account the previous review by Francis

Glassborow nor the comments in Mr Pont's letter, to which I intend to respond in the letters section of this issue.

The author first explains the scope of the book, and it's here that we find out that the title is not entirely representative of the book's content. "Using the Keil C compiler for the 8051-series Microprocessor in Simple Embedded Devices" would be more descriptive of what is actually present. Indeed, there is no focus on teaching C here, not even aspects of the C language or library that are affected by the embedded environment. This may be why the intended audience for the book is programmers familiar with desktop programming in C, C++ or Java. Given this assumption of previous programming background, some of the points made in the book are somewhat basic, and it would be useful for programmers with a mainstream background to be told why it is a sane idea to attempt to produce a time delay with an empty loop that most familiar compilers would optimize away. The author has deliberately concentrated on a single 8-bit processor family, a reasonable choice but it really should be stated more prominently.

This is a book about software, not embedded hardware, though the examples given do include a fair amount of coverage of the physical details of devices the code is intended to control. There is also a considerable amount of coverage of the capabilities of the various devices in the 8051 family of 8-bit microprocessors, and it is here that the book's greatest strength lies. The author makes a point that this is not a book about hardware, and yet there is much background discussion of simple hardware devices. This is useful to provide context for the example code, but the introduction did lead me to expect a book focusing more on the software aspects and less on hardware.

Note that much of the book consists of printouts of code which is on the accompanying CD. This is not a decision which seems wise to me. The code is not particularly readable on paper, being printed entirely in a monospaced font, and there is no reason for any reader to type in out. The code is fairly well commented, but it was disappointing to see the author using names such as _MAIN_H for include guards. Authors should make more of an effort not to write code which is needlessly flirting with undefined behaviour. Too much of the code I see falls into this particular trap [of using reserved names for include guards] and I would have hoped that an experienced author would have warned against it, rather than propagating the bad practice. An explanatory note of why "void main" is used (a permitted extension in C, though not in C++) would be helpful, given that it is common in embedded programming but not on the desktop.

In conclusion: if you are inquisitive about some of the quirks of 8-bit microprocessors and would be interested to learn about one particular family of such processors, then the included Keil compiler and simulator may make this book worthwhile for you. If you are looking for a book offering an overview of how to use C in embedded systems, or a general text covering the huge variety of embedded systems, you should look elsewhere.

**Programming in the OSEK/VDX Environment by Joseph Lemieux (1 57820 081 4), CMP Books, 359pp+CD @ £36-95 (1.35)**
**reviewed by Chris Hills**

[see web]

**Smart Card Manufacturing: A Practical Guide by Haghiri & Taranfino (0 471 4967 3), Wiley, 221pp @ £55-00 (2.45)**
**reviewed by Chris Hills**

[see web]


**Embedded Systems Firmware Demystified by Ed Sutter (1 57820 099 7), CMP, 364pp+CD @ £37-99 (1.31)**
**reviewed by Chris Hills**

The problem with embedded systems is that the only thing they have in common is that they are all different. Running on anything from a 4 to a 128-bit processor, with or without an OS and single or multi tasking.

At the Embedded Systems Show this year (Excel-London, May 2002) I asked the audience at the seminar I was presenting who used what. Whilst the majority were evenly split at 8, 16, and 32-bit, there were a sizeable minority on 64-bit and as many 4 as 128-bit developers in the room. Not scientific but 4-bit is not (quite) dead and 128-bit not so rare.

This particular book is looking at demystifying the techniques used mainly in the 32 and 64-bit end of the market. Though there are also some 8 and 16-bit processor families that also use these techniques. If you want to play with the PICs or 8051 this is not really the book for you.

The main target for the book is the Motorola Coldfire, not the world's most popular chip but the bundled GNU X-Tools on the CD support 20+ platforms and there are ports of the monitor package for 68K, PPC, and SH2 as well as the Coldfire. Also a complete Cygwin, which will give you most Unix utilities running under Win32!

So, having set the scene, what of the book? I found it very good. It works its way logically through from hardware (all embedded systems have an intimate relationship with the hardware) through 'bringing up a board', the initial set up which requires assembly (to create the environment that can call `main`) and then on to programming the flash so that you can put in a monitor to run applications. This is usually done via a JTAG. If you have a target board and a [JTAG] wiggler the book has the rest.

The main area of the book is the building and use of the target monitor. Therefore the reader learns a lot about the monitor, interfacing to hardware, communications protocols, flash programming and creating file systems. In fact most of the basics required for embedded work in the 32-64 bit market.

Students will find this book is a great introduction and lecturers should look at this book for course use. You will need to do more research and read books covering specific topics, e.g. Ethernet, file systems, etc. as this book only really talks you through the example shown. In some cases an area, recursion for example, is skimmed very lightly.

Engineers working in the 16-64 bit market (with JTAG) who spend time with the code on the CD will find it a useful insight, especially if they are using GNU X-Tools or want to develop their own debug system.

I like this book. One last point; the author's example code on the CD has his company and personal email addresses in it. You can't say fairer than that! Recommended.

**Embedded Systems & Computer Architecture by Graham Wilson (0 7506 5064 8), Newnes, 294pp+CD @ £24-99**
**reviewed by Chris Hills**

An interesting book, it is complete and easy to read. A well thought out and structured first book on embedded systems. It covers number basic logic gates, memory, flip-flops, timers cache memory, serial ports, etc.

There are descriptions on how the registers in the processors work with walk thought on things like adding numbers. What is more there is software on the CD that animates the CPU to show the system (and internal busses) working. There is a simple to use logic gate animation and also lots of examples.

It sounds too good to be true. I was about to test it on my teenage son when reality hit. It is all a simulation of a mythical processor. A lot of work has gone into producing an assembler and linker to do with the simulator for this mythical processor.

So do you teach using a real processor or using a mythical one? I have heard the arguments on both sides from many people. Personally I prefer to use a real system (there is usually so much more available like hardware development kits and vast amounts of free code). The problem with using a mythical one is that you get no real world experience and the first thing you have to do is learn a new processor. Assuming you can get a job that is, since you have no experience of a real processor.

The software on the CD installs from a self-extracting `.exe`, so you can't get at any other files without doing a full install. It also insisted on installing to my C drive with no options on directories. On the good side it does check that the PC has the required spec. It is at this point that you discover that it will only run on Win98 or Win 2000! This does NOT impress me. These restrictions should be made clear on the cover of the book, where it loudly proclaims the free CD full of software.

The software is very good for teaching the basics, such as logic gates and internal processor workings, but it's not a real processor type. There are many respected lecturers who do prefer this method.

The book is exactly what it claims to be. An entry-level book for HND courses where this book is the set book. It is of little use to anyone else. (Though this comment does do the high quality of the book a bit of an injustice.) Lecturers at HND level should look at this book if you want to teach techniques without using a real processor.


**Embedded Systems Design by Arnold Berger (1 57820 073 3), CMP, 237pp @ £25-99 (1.34)**
**reviewed by Chris Hills**

The problem with embedded systems is that no two are the same. Therefore a book on embedded systems is either covering a very wide field or looks at a particular aspect of it. Most look at a single development on a specific architecture. This book manages to cover a very wide field but feels as though it is looking at your project!

The text basically goes through an embedded system lifecycle as the subtitle says '*introduction to process tools and techniques'*. Therefore, there is virtually no source code nor a single circuit diagram. There are no hex or ASCII tables nor lists of compiler or development kit vendors.

This book is one of the first I have come across that sensibly describes an almost generic embedded system development. It does not go into enough detail to discuss the use of interrupts and or a specific processor design, but will guide you through the choices and pitfalls of being a project manager or engineer on an embedded system. Which paradoxically does mention interrupts and processor choice. The author has obviously been there and accordingly a lot of the text is anecdotal, but real world, with problems and solutions.

Due to the author having been employed by a silicon vendor, as well as being development engineer and a college lecturer, there are some fascinating insights into the industry that many others will not have picked up and so I enjoyed reading this book.

This is a book that many graduates or new engineers who want to go into embedded work should read. It is difficult to quantify as it gives advice on process and methods without diagrams. One example is development tools such as ICE. The author explains in broad terms what an ICE is and what you can do with it. He then goes on to explain the views and philosophy of development tools as seen from the view of the silicon vendor, the tools vendor and the project management (and accountants). Absolutely fascinating and from my experience, very accurate.

The other main sections are selection process of processor/rtos/tools. The hardware/software partitioning decisions. The development environment, tools, debuggers and testing. The book is not as big as you would think but it's all in there.

So this book will help with the politics of embedded system development. Politics and economics usually intrude into embedded systems far more than any 'normal' development because the embedded system is usually a sub-part of something else. The book will help you to see why the tool/silicon vendor is taking the stance that they are and why sometimes the obvious technical answer may not be the right one.

There are some interesting comments and ideas that could save you a lot more than the cost of this book. The author lists all his references on a per chapter basis, which I found helpful to find source material for a specific aspect of the book. Many are available on the web.

This book has been clearly written by someone who has been there. Useful for students and engineers moving to embedded systems and also for managers. Highly recommended.

**Embedded System Design by Frank Vahid/Tony Givargis (0 471 38678 2), Wiley, 324pp @ £24-95**
**reviewed by Chris Hills**

[see web]

# Software Development


**The UML Profile for Framework Architectures by Marcus Fontoura et al. (0 201 67518 8), Addison Wesley, 228pp @ £33-99 (1.32)**
**reviewed by Silvia de Beer**

A UML profile is a modification of the UML standard to target a specific application domain. A framework is defined as an extensible semi-finished piece of software. The authors propose the UML Profile for Frameworks, which they call UML-F profile in short. The book does not target a large audience of mainstream software developers, though it should be interesting for software designers and architects who want to reflect on the topic of frameworks and the use of UML in design concepts.

The book contains seven chapters. The first chapter defines terms like framework and profile. It explains in what ways a framework can be extensible and states the goals of the UML-F profile. The second chapter discusses the UML notation in the light of the extensions by the UML-F profile. Chapter three explains tagged values and stereotypes and introduces two UML-F specific presentation tags, the completeness and incompleteness tag. Inheritance indicators are introduced to be able to indicate in class and object diagrams whether methods and attributes are abstract or can be overridden. Some new tags are introduced for the sequence diagrams to indicate triggers and repetition of messages. Three other tags for class diagrams are used to indicate which classes belong to the framework and which classes to the application. The `<<fixed>>`, `<<adapt-static>>` and `<<adapt-dyn>>` tags are introduced to indicate how classes and methods can be specialised by an application, e.g. a fixed tag does not allow overriding of a method in application classes.

Chapter four presents framework construction principles and introduces the corresponding UML-F tags. Two important construction principles are discussed, the unification and separation construction principle. Template and hook methods are explained. At the end of this chapter it is shown how UML-F tags can be defined for the Composite pattern. Appendix B defines more tags for the GOF patterns. Chapter five discusses how to facilitate framework adaptations, i.e. guide the writing of real applications and discusses how a cookbook can consist of a guided tour and a collection of recipes. Some example recipes are given.

Chapter six provides a case study that exemplifies the theory of the previous chapters. The last chapter gives some hints to framework development, which in itself is more complicated then developing an application according to given requirements.

I did not agree with all of the statements made about framework development, but on the whole this book is an interesting and original piece of work, presenting lots of useful concepts for framework design.


**Making Process Improvement Work by Neil Potter & Mary Sakry (0 201 77577 8), Addison-Wesley, 169pp @ £22-99 (1.30)**
**reviewed by Pete Goodliffe**

Most software development companies know that they could be doing things better. Many have 'process improvement' programmes to work out how to do just that. Only a few of these will have any appreciable effects.

This book provides insight into how to improve software processes. It doesn't labour why you'd want to do this, which is a benefit – if you're reading the book, you know the why already. It is a concise, practical guide aimed at the people directly involved in implementing the improvement process.

It is well thought out and sympathetically laid out. Comprising just three chapters, the book jumps pretty much straight in to the deep end without waffle. Starting with 'Developing a plan' the authors suggest practical and pragmatic ways to identify areas to improve and also suggest how to 'sell' the improvement process to co-workers. The 'Implementing the Plan' and 'Checking Progress' chapters follow on logically.

This is a well-written book, the authors certainly convey their wealth of experience. There are plenty of examples, but they are not taken overboard. The level of detail is pitched carefully and the book defers to other texts in many places where the material is not core to the main thrust of the discussion.

In fact most of book is done by page 115. The rest is a set of (useful) appendixes, which contain subject matter that was pulled out of the main body of chapters to prevent breaking up the flow.

Although potentially appealing to only a restricted number of readers, if this is a subject that you need to read about, this is an excellent reference.

**Tricks of the Windows Game Programming Gurus 2ed by Andre LaMothe (0 672 32369 9), SAMS, 1063pp+CD @ £43-99 (1.36)**
**reviewed by Francis Glassborow**

[see web]


**Questioning Extreme Programming by Pete McBreen (0 201 84457 5), Addison Wesley, 199pp @ £22-99 (1.30)**
**reviewed by Francis Glassborow**

All the other books on XP that I have read, or browsed as they crossed my desk on the way to other reviewers have been seeking to convert the reader to the joys of XP in some way or other. This book is different in that it seeks to question all aspects of XP. In a way this shows the degree to which XP has taken root. Pete McBreen seeks to challenge XP whilst respecting its objectives. Kent Beck (the creator of XP) says this in the foreword he provided:

> Pete claims that the more he looks at XP, the smaller he sees its scope. I see just the opposite. I won't refute his argument point by point– this is a foreword and I'm supposed to be polite. I will suggest that as you read this, you keep in mind one mistake of early XP thinking for which I am entirely responsible – "the customer" doesn't mean *one person*. It means a team, as big as or bigger than the development

Part I of the book consist of a single short chapter in which the author takes an overview of the claims of XP, the evidence for its success and reasons why you might adopt it.

Part II consists of five chapters that are largely aimed at reminding the reader of the alternatives as well as the aims of having a methodology at all. He spends a fair amount of time reminding the reader that XP is only one of a group of methodologies that have come together as 'the Agile Alliance.' Those that have not previously come across this should take time out to read a book such as the excellent *Agile Software Development* by Alistair Cockburn (0 201 699 69 9).

The author then moves on to the main parts of the book (III & IV) in which he examines XP in detail. Here he suggests things that other methodologies could learn form XP as well as how XP has itself changed and learnt from experience. As I was reading these sections it crossed my mind that there is a degree of irony in the XP adherent who considers XP to be the one true way; one important point of agile methodologies is to be able to adapt to the needs of the moment and that means you should recognise when an XP doctrine is inappropriate. This book will help those who do not have entirely closed minds to understand that. If you cannot recognise that having your beliefs challenged makes you beyond salvation then this book will do nothing for you whether you are attracted to or repulsed by XP.

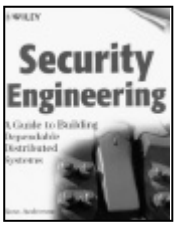
Part V of the book concerns understanding the XP community. Note that this is relevant to both the outsider looking in and the insider. Perhaps it is more important for the latter because it reminds them that there is a world outside the closed development team in which they work. One very interesting chapter in this section is chapter 21 *Transitioning Away From Extreme Programming*. Here the author tackles the problem of how to break XP devotees away from it. It is a short chapter but contains food for thought. Here is what the author writes early on:

> This is an interesting question, because XP is very appealing to programmers but is not suitable for all kinds of projects. Yes, it may be feasible to restructure your organization and projects to make them more suitable and amenable to Extreme Programming, but for some projects you may have to find an alternative process. In doing so, you are likely to face resistance from your XP teams because the sense of control over their work that Extreme Programming gives is very addictive. Very few people who have worked as part of a well-functioning XP team have expressed any interest in using a different process.

The book concludes with Part VI that consists of two short chapters giving guidance as to whether XP might suit your needs, and how to choose a project to start XP on.

I think this is a book that deserves to be widely read by those whose minds are open enough to make choices based on reason rather than emotion. Agile development is much liked by programmers because it makes the process closer to the way their instincts say it should be. XP in particular is a methodology that takes over the thinking processes of its adherents and too often blinds them to the essential purposes of a methodology. The outsider looks on and sees something akin to religious fanaticism, the insider cannot see how anyone could not appreciate how good XP is. For the devotee, failure must mean lack of sufficient devotion and for the antagonist success is just coincidental. This is a book from which programmers of goodwill can learn and in doing so apply the lessons to what they actually do. This book is an excellent read and food for much thought.

## General Computing




**Security Engineering by Ross Anderson (0 471 38922 6), Wiley & Sons, 602pp @ £42-95 (1.40)**
**reviewed by James Amor**

Ross Anderson is one of the most respected authors in the field of computer security and this book represents the culmination of his experience as a security consultant. Anderson takes a fresh approach to documenting security engineering; the contents pages are not simply a listing of cryptographic algorithms, as found in so many other titles, but a myriad of subjects ranging from biometrics and intrusion detection, through to management issues and system evaluation. Anderson first introduces the basic concepts of security engineering, next the application of these concepts are explained and finally various management issues are covered.

This book is suitable for all readers; whether new to security engineering, or a seasoned professional, everyone is guaranteed to learn something new. The explanations of all concepts are excellent, with fascinating case studies littering the book. Personally I found the entire book fascinating; with topics ranging from nuclear command and control, ATM cash machine defence and physical smart card protection, each chapter stimulates the reader to consider the security of many everyday objects.

I cannot recommend this book highly enough and in my opinion, every computer professional should have a copy on their bookshelf, for anyone whose interest includes security engineering this is essential reading.


**T1 A Survival Guide by Matthew Gast (0 596 00127 4), O'Reilly, 290pp @ 20-95 (1.43)**
**reviewed by Mark Easterbrook**

If you need to know T1 for data networking in detail, even infrequently, then this book belongs on your bookshelf. In reading the book from cover to cover I couldn't spot anything obviously missing.

However, it will be a must-have book for only a small number of engineers and that number is diminishing each year. The intended audience is described in the preface; *This book maintains a blatant and sometimes overwhelming, focus on the U.S. This book approaches T1 from the narrow perspective of data networking*. This is not surprising as T1 is only used in North America and a small number of Pacific Rim countries _ it has little information for readers outside those territories, nor for the voice telecomm engineer. Even in the U.S. the 1.5Mbit/s T1 as a medium-speed data access transport is being displaced by newer and cheaper technologies such as cable, xDSL and terrestrial radio and satellite.

The book does contain some subjects with wider appeal, such as the chapters on HDLC, PPP and frame-relay, but it is not worth purchasing the book just for these.


**Site-Seeing by Luke Wroblewski (0 7645 3674 5), Hungry Minds (Wiley), 341pp @ £34-99 (1.43)**
**reviewed by Francis Glassborow**

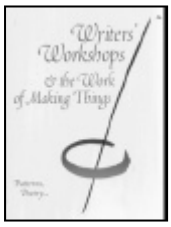
I am going to keep this brief, no more than drawing your attention to this book, because it has nothing to do with programming other than that some programming is used in making websites.

The book's subtitle, *A visual approach to web usability*, says most of what is necessary. The author has written a carefully considered text on website design with copious full cover illustrations from existing sites. The strong underlying theme is that it is all about appropriate communication. If you are involved in designing a website (yes, of course you are because in the present age you have a home page - actually I do not because I lack the time to design a good one and would not put my name to a bad one) this is one of the books you should consider reading before you start.

**Operating System Concepts 6ed by Silberschatz et al (0 471 26272 2), Wiley, 949pp @ £31-95 (3.02)**
**reviewed by Francis Glassborow**
[see web]

**The LATEX Companions by Goossens, Lamport et al. (0 201 77591 3), Addison-Wesley, 4bkspp @ £106-99 (1.30)**
**reveiwed by Francis Glassborow**
[see web]

# Non-Computing?




**Writers' Workshops & the Work of Making Things by Richard Gabriel (0 201 72183 X), Addison-Wesley, 268pp @ £22-99 (1.52)**
**review by Francis Glassborow**

When this title landed on my doormat my first instinct was that it was one of those books that has been sent to me by accident. I bet your instinctive reaction is that poetry and programming have little to do with each other. If you stop for a few moments and think about it you may realise that they actually share a very great deal. Both programmers and poets create by writing in a language that is constrained by a bundle of rules and guiding principles. There is a tendency for both to work alone. Both groups need exposure to the thinking of others.

Of course there is more. We need to learn to write documentation, both the specific for an application or library or the more general such as researching and documenting a software pattern.

Now there is poetry in the souls of many programmers. I am sure that is one reason that some revolt against such things as Hungarian Notation (it looks so ugly). We do not just want to write successful code but we want to write elegant code. Good code does not just appeal to our intellects but to our aesthetic sense as well. I wonder how many poets, romantic novelists etc. would consider that they had anything in common with programmers.

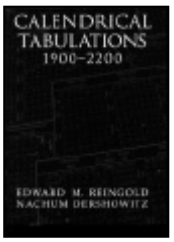
Now this little book addresses the wide range of issues concerned with organising and participating in successful workshops for writing regardless as to what sort of writing is involved. Let me quote the first paragraph of chapter 5, *The Gift*:

> The writers' workshop begins with some people's decision to give each other the gift of their work in progress, and a more experienced individual's decision to give the gift of experience and expertise as a workshop leader. The magic of the gift.
>
> The magic of the gift is not something new – it's always been part of human culture. The gift economy has been studied deeply. It's how our families are held together. It's how many ancient and contemporary cultures are held together. It forms the center of many religions. The writers' workshop works best when it is most firmly based on a gift economy.

Now does that seem familiar to you? If you are a member of ACCU it should because that is one of the fundamentals on which it is built. It finds expression in many ways, those who take time to write for our journals, those who participate in our Mentored Developers programme and those who participate in our conferences. Indeed reflecting on much that this author writes makes me realise that the success of our conferences is exactly because we have instinctively created something that is closer to a workshop than most technical conferences.

I would encourage you to read this book, to think about the message and then think how you can apply it both in your place of work and within ACCU. Perhaps it is time that we thought about having summer workshops. Read this book and let me, or better still your committee know what you think.

Now for non-ACCU members, I think this is an excellent book in that it gives guidance on one more way that your professional development can progress. I think that I would make this book compulsory reading for anyone organising a workshop. I would also make it highly recommended reading for anyone planning to participate in a workshop. Yes, programmers are writers, and they do make things and some even have poetry in their souls.




**Calendrical Tabulations 1900-2200 by Edward Reingold & Nachum Dershowitz (0 521 782 53 8), CUP, 605pp @ £85-00 (1.41)**
**reviewed by Francis Glassborow**

This volume takes me back to the time when it was normal to have solid books of tables in which to look up seven figure logarithms, sines etc. Here is the first paragraph of the preface:

> We give tables for easy conversion of fifteen different calendars. Ten calendars are given explicitly (Gregorian, ISO, Hebrew, Chinese, Coptic, Ethiopic, astronomical Persian, Hindu lunar, Hindu solar, and astronomical Islamic); another five are easily obtained from the tables with minimal arithmetic (JD, R.D., Julian, arithmetical Persian, and arithmetical Islamic). Detailed explanations of the structure and determination of these and many other calendars can be found in [10].

The reference is to the authors' book *Calendrical Calculations*.

In the next paragraph they justify the book on the basis that there are no existing computer programs that handle more than a couple of calendars. I think this book is the wrong response to this. What is needed is a good program that can be run on a palm-top. Such a program would not be limited to just three centuries and so would be useful to people who need to convert between historical dates. I would not expect there to be a very big market for this book despite its quality and coverage of all calendars currently in use. There are a few people who could benefit from copies sitting on their reference shelves. A Daily Mirror journalist, for example, might have saved himself the embarrassment of assuming that the current date in Afghanistan was an Islamic one (they use the astronomical Persian calendar)

Note that the computation times given by the authors show that on the fly computation can be done for most calendars. The exception being the Hindu Lunar calendar which requires some very complex computations and would probably require noticeable computation time on the current generation of palm-tops.

I suspect that much of the quoted computation time (about 10 minutes per year) was the result of not only computing the dates but also the page layout.

An interesting book, but, sadly, even libraries should think carefully about its value. Now anyone willing to produce a computer program to calculate dates on the same range of calendars but starting much earlier. Let me give you an example. Someone studying the crusades might get real insights if they could understand how Islamic religious dates interacted with Christian ones.

# Reviewer Needed


**Real-Time Interactive 3D Games (uses Macromedia Director by Allen Partridge (0 672 32285 4), SAMS, 604pp+CD @ 43-99 (1.36)**
**request for reviewer**

The sub-title is *Creating 3D Games in Macromedia Director 8.5 Shockwave Studio*, which really means that a reviewer needs some familiarity with this product. If you meet that qualification and the following from the back cover attracts you trying the book out, please let me know and I will pass the review copy on to you.

> This book provides an exciting blend of game design, nuts-and-bolts programming tips, deeply detailed code samples, and richly developed games and demos. *Real-Time Interactive 3D Games* defines the role of narratives in games and is packed with valuable advice on a variety of topics, including tapping into the traditions of theatre to make the games you develop more dramatic and tips for marketing your 3D game.