Contents

Reports & Opinions

Reports

Dialogue

Features

Python

Reviews

Copy Dates

C Vu 15.1: January 7th

C Vu 15.1: February 21st **(Note early copy date, to enable publication of the April issue before the Spring Conference)**

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

Should an editorial for December start by wishing you all a good Christmas and New Year? Only if I also offer wishes for those celebrating other cultural events, so I shall do that. For those in the US, I offer my apologies for not having wished you a good Thanksgiving break, and for those in Canada I apologize for not knowing when your Thanksgiving break **is**. (Note for those in the UK: pumpkin pie tastes better than it sounds.)

Education and Professionalism, Continued

In my previous editorial, I wrote a few words about the way programming is taught in our colleges. Now I would like to step back a little to consider what the relationship between students, educational establishments and industry should be.

There is a long-standing debate over whether education should be for the benefit of the student or for the benefit of the provider of that education (which may well, at least in large part, be the state). As with so many apparent dichotomies, we should be wary of allowing our thinking to be constrained by the question. Firstly, beware of being pushed into black-and-white choices: I do not expect to hear significant dissent when I say that education should benefit both the individual and the state to some level, so that the question at most is where the balance should be. Secondly, consider the image which arises from thinking in terms of a trade-off between value to the individual and value to the state: we think in terms of a linear model where movement towards one end is necessarily movement away from the other. If, instead, we think in terms of two perpendicular axes, it is easier to consider factors which benefit both parties. The linear view breaks down in any case when we factor in the third party in this case: industry, who are in a sense consumers of the results of the educational process.

With all this said, back to the question of what our education system ought to aim to provide for the three stakeholders we have identified (roughly: students, state, industry). Should colleges be churning out project-ready software engineers? Many in industry seem to think so, and complain that the graduates they see are not ready for "real work". (Publications such as "Computing", driven by the IT industry, often print such views.) Individuals sometimes complain that their courses don't provide them with the up to date skills requested by industry – and in the current economic climate that is understandable, as companies are even more reluctant than normal to employ people who need training when there are so many experienced practitioners seeking work. Should we switch to a more practical (applied, specific) style of education in the field of computing? Industry wants it, students want it... so why not?

Before discussing why not (all in time!), let us ensure that we are not slipping into black-and-white thinking again. The simplistic question is which direction to go in: vocational or theoretical. As usual, some combination may be best, and that combination may not be a simple question of proportions. Diversity is so often a good thing, and in this case it could be reflected by a combination of different approaches: some technical colleges could choose to offer courses with a relatively practical emphasis, while traditional universities might prefer a focus on "computer science", whether or not they agree with the opinion that sciences (say, physics, biology, chemistry) are distinguished by the fact that their names do **not** include the word "science". So, allow us to think in terms of a diverse selection of courses to suit different goals and different kinds of people.

Enough of this whole balanced viewpoint exercise. I would like to argue. Specifically, I would like to argue that there is more merit in theoretical computer science courses than is perceived by industry; to put it another way, that what industry **wants** is not necessarily what industry **needs**. Industry cries out for production-ready engineers dropping from universities which are there so that industry does not have to pay for more costly training courses, or develop the capability to develop the skills of its workers. Even if we assume for a moment that the educational establishment could provide these droids, it would not solve the major educational problem facing our industry. That problem is that industry either does not recognise the value of continual education or, as with so many other things, knows what it ought to do and yet continues to concoct excuses why it cannot do it. Companies are reluctant to invest in training because they fear that they will not see the value from it, as employees leave to join other companies. Most of us do not actively look forward to searching for a new job, so it is odd that employers should live in fear of staff moving away – and yet, maybe slightly ironically, the dissatisfaction that leads employees to seek greener pastures often arises from the same thing that stifles opportunities for training: deadlines. A discussion of what software professionals can do to improve that situation will have to await a future editorial. For now, back to education and its relationship with industry.

For one easy point, it is clear that an educational course in computing cannot cover the whole field in any depth. Even if we restrict ourselves to a good grounding in general computing concepts combined with a practical knowledge of a small representative selection of programming langauges, operating environments, tools etc., there is too much to cover in a single degree 3- or 4-year course. That means – just as in industry – that compromises must be made, some important items must be omitted and others must be covered only in overview. Debate over what to cover should not focus on whether the knowledge is important or not, but rather with whether adding them to a course adds more than would have to be lost by taking other material away to make room. There is a scale invariance to this as well; for each subject that is included, it is necessary to choose which parts to present. (I am trying hard here not to call them sub-subjects; there is a limit to how much abuse the English language should be asked to endure). To take an example, a Computer Science degree might include a one term course on C++, and then will need to choose which parts of this expansive language (including its library) to cover. That is emphatically not a case of saying that there are areas which should be ignored or covered up, but rather of choosing how to get the "best" combination of material covered. What is "best" depends on the goals of the course. For students of electronics, best might mean learning just some of the C-compatible subset of C++, or sticking to C, or even eschewing the abstractions of C and C++ and learning assembler of some form. For those who plan to be pure software engineers, it may well be most valuable to focus in a first C++ course on what its advocates refer to as "modern C++": C++ making extensive use of the standard library, using exception-safe forms, and aiming for consistency with the idioms used in the standard library. For colleges which choose to teach C++ as a first language in an introductory course for non-CS students, simple use of standard library facilities (such as preferring `std::string` to raw `char` arrays) can help students to understand the ideas of programming without having to have an in-depth feel for the underlying machine. All of these choices (and others besides) are reasonable, so long as the goal of a course is clearly stated. One thing a college C or C++ course will not do is to turn a novice into a software engineer capable of authoring production quality code. Colleges should not aim to do so, and employers should not expect that from fresh graduates; prefer those who know their own capabilities and limitations to those who have a command only of contemporary (fashionable?) tools. This is an industry in which continual learning is essential throughout our careers, and where the best in the business delight in learning from others.

Enough for now; this is a subject which can run and run.

Journal Composition

I want to say a few words about C Vu (apparently that's allowed in an editorial piece). Some have said that they consider C Vu as being restricted to the C language only; Overload, they say, is for C++. And certainly Overload is the main focus of ACCU's printed C++ material, but the division between the two journals is not quite so simple. Overload has its focus, and while it is not my place to describe that I will say that it tends to cover more in-depth pieces on more advanced subjects. C Vu is more general; it's here to publish whatever the membership want to see, so long as the membership are willing to write it. If more C material is wanted, then consider writing it. Similarly for C++. Or if you don't feel that you can write yourself, then tell me what you'd like to see; somewhere among the membership of ACCU there is likely to be someone who could write an article on it.

James Dennett

From the Chair

Alan Griffiths `<chair@accu.org>`

Every two months I receive a reminder from the C Vu editor that it is time to write to you about what is happening in the ACCU and over my term there has been much to write about. It is with a certain

sense of relief that I find that there is little going on in the organisation to tell you about.

But that very sense of relief is a cause for concern. When I took on the job I thought that the biggest challenge would be to find ways to make contact with the many developers who should be members of the ACCU (both for their benefit and that of the organisation). I should be viewing the lack of disasters or initiatives to announce as an opportunity to further this agenda. I should, but I no longer do. I conclude that it is time for me to move on and let a fresh mind address the problem of chairing the ACCU.

I hope that I've achieved something worthwhile as chair. My term has been a period of significant changes to the organisation: both with changes to the production of the journals, and with the withdrawal of Francis Glassborow whose energy had been the driving force behind ACCU for as long as I could remember.

I've got a tremendous amount out of the ACCU and from being Chair, but I need to do something different and you need a Chair with the enthusiasm to take the organisation forward. I still want to contribute, but in some other capacity. I don't intend to accept nomination for Chair at the 2003 AGM.

Membership

David Hodge <membership@accu.org>

Total paid up membership is 970. New members continue to come in at about 20 a month. From now on you will not be able to join or renew by standing order from July-October. Standing orders can still be set up from November to June, but from the next August 15th onwards. The reason for this is that setting up standing orders that have a start date and an August the 15th date is something the banks don't seem to be able to cope with. So if anybody wants to start a standing order for next August 15th (2003) then just email me for a form.

Conference Programme

Francis Glassborow
<francis.glassborow@ntlworld.com>

The main programme is now complete. Well almost because there are still a couple of important speakers that are unable to confirm that they will be able to come.

Unfortunately, despite being the one speaker most asked for, Scott Meyers is unable to make it. However when you see the overall line-up most of you will be relieved that you do not have to make yet another hard choice. I have no doubt that this year's conference will raise ACCU's already high standards.

I know that cost of attendance is always an important consideration for some. I also realise that the cost might be reduced (by about 15%) if we had only a 1-track event because the ratio of speakers to attendees would be more favourable. Last year the ratio was about 1:5. However many speakers do not claim all their expenses and many come exactly because they enjoy the event and also enjoy meeting other speakers as well as the attendees.

I have no doubt that the ACCU Conferences are among the best in the world but that message needs spreading so that more employers are willing to pay the costs of attendance as well as a larger overall attendance reducing the costs.

Please consider whether you should not only be coming but also dipping your toes in the water by offering a 30-minute lunchtime presentation. This would be an excellent way to get started on the conference circuit and we will try to give new speakers feedback on how they can improve (assuming that they are not already world class)

I intend that this will be my last event as Conference Programme Organiser so I hope that you will be among those who come and make it an outstanding success.

Standards Report

Lois Goldthwaite
<standards@accu.org>

The international C++ Standards Committee met in Santa Cruz late in October. The UK was represented by Francis Glassborow, James Dennett, Jason Merrill, and Lois Goldthwaite. Other participants came from the US, Denmark, Japan, Canada, France, Norway, and Germany.

The Library sub-group of WG21 (official name of the C++ committee) is looking toward the next revision of the Standard and what new functionality should be added to the Standard Library, with any additions to be based on well-tested existing practice. Much of the development of existing practice is coming from the Boost group (`www.boost.org`). Two of the Boost libraries — Tuples and Polymorphic Function Wrappers — were accepted for inclusion into the Technical Report which will lay the groundwork for the next Standard. Another proposal for Regular Expression Objects, from the UK's John Maddock, will almost certainly be included as well.

Speaking of C++ Committee Technical Reports, the one on C++ Performance is nearing completion. You can read the latest draft at `http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/papers/2002/n1396.pdf` (where you can also find papers on the topics in the previous paragraph — look for `n1402.html`, `n1403.pdf`, and `n1386.htm`). ACCU Standards Officer Lois Goldthwaite[1] has recently been named as chair and editor for the Performance sub-group, so if you read this paper please send your comments to `standards@accu.org`.

Software is becoming increasingly pervasive in everything around us — automobiles, washing machines, children's toys — and therefore everyone should be concerned about whether this software is reliable, especially when safety is at stake. Carnegie Mellon University in Pittsburgh, Pennsylvania, along with a couple of dozen business and government organisations like NASA, Oracle, Microsoft, FedEx, and Surrey's Programming Research, have founded a group called the Sustainable Computing Consortium (`www.sustainablecomputing.org`). The objectives of the group are to drive order-of-magnitude improvements in software quality, dependability, and security. They say recent estimates suggest that defective software accounts for 45% of computer downtime and cost U.S. companies alone over $100 billion annually. Carnegie Mellon is the home of the Computer Emergency Response Team (CERT), which gathers information on internet security problems, and the originator of the Capability Maturity Model for evaluating the process of software development.

SCC are developing standards and specifications, plus ways to measure and reduce risk, both that associated with software and also risk to organizations, the broader markets, and the economy. (Think how much of the international infrastructure in communications and transportation is dependent on software.) In addition to a series of metrics (the "Sustainability Index") to quantify quality, dependability, availability, security and survivability, they will be producing tools and technologies to support these improvements. Programming Research, which markets software to audit software for standards compliance and safety-critical good practice, has a natural alignment of interest with SCC.

The aims of SCC are highly laudable, but there is also controversy associated with the project. The joining fee is $25,000, with a $5000 concession rate for government and non-profit organisations. Furthermore, membership only confers a license for internal use of the intellectual property created by the SCC; developers will have to pay royalties if their products incorporate technology created by the consortium. Open Source developers are hoping CMU will reconsider the emphasis on proprietary solutions (and some of them argue that it is closed-source, proprietary software that has created such a big problem in the first place). The American space agency NASA has already donated $23 million to CMU to pursue studies in high-dependability computing. Earlier, they underwrote the FlightLinux project (`flightlinux.gsfc.nasa.gov`) to develop an embedded real-time operating system for use on spacecraft, but that project's original period of funding expired last summer. If NASA are turning away from Open Source, Tux may never get a ride into space.

Mentored Developers

SI Units Project

Pete Klier <pete.klier@lmco.com>

The SI units project has made significant progress up to the point of near completion. The requirements have been essentially completed and an implementation has been provided by Simon Watts, which includes the usage of metaprogramming and the capability to handle rational powers. Thus the basic skeleton can be considered to be complete.

The challenges ahead of us include the incorporation into the above system of the ability to use baseline types in a fully general way: For instance, if `T op U` yields `V`, then `velocity<T> op velocity<U>` should yield `velocity<V>` if it is otherwise legal. A draft implementation is being peer-reviewed.

Beyond that, we would like to incorporate good compile-time error messages, as well as the ability to print types in a "smart" way, i.e. recognizing "ohms per metre" as such without a prior declaration of same.

Progress has been hampered by the fact that nobody seems to have much time on their hands due to annoying things like actual jobs. However, it would be nice to finish up the above, make certain that the system is user-friendly and well-tested, and then perform some sort of write-up or publication if we haven't been trumped already – however, we are implementing a stringent set of requirements and thus we hope that this effort will be useful to the community.

1 I recently discovered that a person who refers to him- or herself in the third person is termed an 'illeist'. There is a word for everything if you look hard enough!

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.

Student Code Critique 18: the entries

First a reminder of the code being critiqued.
I am trying to compile the following code and getting an 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
```

From James Holland <jamie_holland@lineone.net>

The compiler error message quoted in C Vu is somewhat curious. The message refers to the header declaration of `operator==()` as having a reference parameter. The header (`pgsimeta.h`) makes no mention of a reference parameter. Is this a typo in the text of the problem? The parameters of the declaration and the definition of `operator==()` must match. They must either be value parameters or reference parameters. The `operator==()` function does not need a copy of the `PgsiMeta` object passed to it and so it would be sensible to refer to the object by reference. In other words the declaration of `operator==()` should be `bool operator==(const PgsiMeta &);`

The error message states that `operator==()` must take exactly two arguments. Now, if `operator==()` were to be implemented as a non-member function it would require two arguments (the first argument as a reference to the object on the left of the `==` operator and the second as a reference to the object on the right). The student has declared `operator==()` as a member function and so only requires one argument as a reference to the object on the right of the `==` operator. The compiler has clearly got itself confused.

In any case, my compiler does not issue the same error message as the student's. The messages it does come up with are just as unhelpful, though. It's a pity but the C++ standard does not require compilers to be helpful!

The first thing my compiler complains about is that the `typedef` in the header is missing a semicolon. The semicolon plainly exists at the end if the line. The compiler is definitely confused. This is quite typical. You should not take what the compiler says too literally. It has definitely found a problem with `typedef` statement, but what? This is the sort of problem that can keep the programmer scratching his head for ages.

It turns out that the class `vector` has been defined within a namespace with the name of `std`. The compiler has no idea what `vector` is because the compiler is not looking in the `std` namespace. The compiler can be made to look in the `std` namespace for its search for `vector` by preceding the reference to `vector` by the namespace name `std`. The statement in question now becomes `typedef std::vector<MetaData> DataList;` After correcting this problem we can recompile the program and see what the compiler has to say for itself this time.

The compiler is now complaining about invalid use of namespace `PGSIMeta` in the second line of `pgsimeta.cpp`. This time the compiler is spot on with its diagnosis. The statement `using PGSIMeta;` is simply incorrect. What is required is either `using PGSIMeta::PgsiMeta;` or `using namespace PGSIMeta;`.

The using-declaration `using PGSIMeta::PgsiMeta;` instructs the compiler to look in the `PGSIMeta` namespace for the identifier `PgsiMeta`. The using directive `using namespace PGSIMeta;` instructs the compiler to include the `PGSIMeta` namespace when it is looking for any identifier. Either way the compiler should now find the definition of `PgsiMeta`. There is another way to solve this problem and that is to add the namespace name to the definition of `PgsiMeta::operator==()`. The definition of the equality operator would now become `bool PGSIMeta::PgsiMeta::operator==(...` A similar thing was done when the namespace `std` was added to `vector` above.

The program should now compile without error and produce the correct result. There is one enhancement that could be made and that is to make `operator==()` a constant function. The function does not alter the value of any member variables and making the function constant will make that explicit.

For completeness I provide a copy of the corrected program.
In file `pgsimeta.h`

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

In file `pgsimeta.cpp`

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

James

Note From Francis Glassborow

I want to raise a couple of issues with the above critique that demonstrate two common misunderstandings. The first is that if your compiler compiles it the way you expect then everything is correct. Compilers do not define C++; only the Standard does that.

The second is that using declarations and directives do the same kind of thing. Using declarations make those declarations behave as if they had been declared in the declarative scope of the using declaration. Using directives on the other hand tell the compiler of another scope to search for declarations.

Nonetheless I would prefer the style of reopening the namespace as being less problematic:

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

Entry from Simon Sebright

<simonsebright@brightsoftware.freeserve.co.uk>

Here's my entry. Edit the waffle if you like, but I notice that most ACCU articles take a sentence or two to get down to business. I have assumed that the code given is the entire content of the two files. If not, then some of my points won't be valid...

As John Denver would say in the modern age, the combi boiler's gently pumping water round the radiators, and pasta's on the gas hob. And of course, there's some beer on the desk. A homely environment for our student. If we meet outside the learning centre, perhaps a level of mutual respect is maintained. One would hope that this would lend more credence to any arguments put forward. As this is a monologue, I'll have to assume answers, or allow for various options.

Well, your code doesn't compile, huh? Let's see. Yes, rather a long error message, I'm not sure I follow it either. My compiler (VC6) complained about an overloaded function not being found in class `PgsiMeta`. And true enough it isn't. I spotted this quickly, you know. Not because I'm a human compiler, but because I found your declaration of `operator ==` rather odd.

Is there any reason you chose not to use the canonical form? That's the usual way of doing it. Have a `const` reference as your thing to compare against. When do you think it's best to use a by-value parameter, then?

Bit of a discussion ensues in which we boil down to builtin types. Oh, and it's usual to call the parameter to `operator==`, copy constructors, etc. something like `other`, `rhs` to indicate the relationship to the `this` object. `obj` is bland, says nothing other than that you had to give it a name in the implementation to get it to compile (yes, I note there isn't a name in the declaration, why's that?).

Does this function modify a `PgsiMeta`? So... Yes, make it `const`. I personally prefer to have to ask myself the question, "Should this **not** be `const`?". Would you modify either side in a comparison? Isn't it symmetrical? Right, the parameter is `const`, so should `this` be.

Why is your destructor virtual? No other virtual functions. It's not pure. Ah, it was wizard-generated code. Hmm. And you trust that to know the ins and outs of your class, do you? Or, ah, you read somewhere about the dangers of non-virtual destructors. Here's a thought, you make it non-virtual and think how you would answer the opposite question. Hint, read some Scott Meyers.

That's not an issue about compilation. Neither is this question. Why did you feel the need to comment the fact that `class MetaData` is defined at the top? Firstly, it isn't. Secondly, for the code to compile, it must be defined (prior to this point in the pre-processed file), so anyone looking at your code would know that. It's completely meaningless, and as the only comment in the file, it shows you don't understand the purpose of comments.

[Actually, I think that in this case the comment was simply to avoid having to provide a lot of source code that was not germane to the problem, or at least did not appear to be. Francis]

Where's the comment describing what `PgsiMeta` does?

Compilation issue – your using statement is missing the keyword `namespace` – `using namespace PGSIMeta;`

Stop, before you do that, why? Yes, you get the remaining code to compile, but I would contest that you are not using the namespace when you define the functions of `PgsiMeta`, but defining them. Yes, deliberately tautological.

Scrap it.

Well, commentless, we don't know what it's all for. What's the meaning of life for this class? Why is it in a namespace? Not that it shouldn't be, I just want to know why you put it in there. Why did you make the name of the namespace and the name of the class the same, bar case? One class per namespace, or is it because you read about namespaces and thought you have to have one and couldn't think of a name? You need to think about these things, you know. Without looking, write down the names of your namespace and your class. Quick.

Where are your constructor and destructor?

Here's a question – do you need your `operator ==` at all? What does the compiler do if you don't specify one? Go and find out. This is a teacher's trick, as off the top of my head, I'm not sure if it'll do a bitwise comparison or a member-by-member comparison, the latter obviously being fine for our purposes.

[Actually it will generate an error as operator== is not one of the functions that compilers are permitted to generate if you do not provide one. Francis]

There's no divine solution here. That's mainly because I can't see the purpose of this class other than as student testing something, and what that is is not even clear. With no public or protected members other than constructor and destructor, it's pretty devoid of use. So, prefix all of my entry with, "What are you trying to do?".

Simon

General Comments from Francis Glassborow

I knew this was a tougher problem than it looked. I also suspect that quite a few readers skipped over it on the basis that it was too simple to try. Or perhaps some spent time staring at it and wondered how it generated just the single error message when so obviously there was an earlier place where any quality compiler would call you out.

We generally ask that when faced with an error that you do not understand, strip the code down to the bare minimum that exhibits the problem. I think this is clearly what the student did this time, so I would make a quick comment on the issue of `MetaData`, I would also assume that he had in fact handled the problem of `vector` being in namespace `std`. If he had not his code would not have compiled.

The two main issues are a design one and a coding one. A third, and for me minor one is the naming conventions. And lastly is the issue of whether the class should have a virtual destructor. That last one can only be handled if the exact intention of the whole class is known. As the student has clearly stripped out all but the bare minimum to show the problem I think that nothing more than the mildest comment on the destructor is called for.

The design issue concerns the choice of parameters for `operator==`. As this operator is being provided as a member function I am pretty sure that should be a const member function and take a const reference for its parameter. However that leaves me a little worried if this is a base class, because it allows conversions on the right-hand operand though not the left.

The syntax issue, the one that was responsible for the somewhat unhelpful (English understatement) error message, is interesting because both entries have missed the real requirement whilst identifying the cause of the error message.

The general rule is that definitions must always be in the same scope as the declarations. All the forms of `using` are about making something visible for the purposes of name look-up. But they do not do the same kind of thing. I would strongly advocate either the solution I suggested earlier or:

```
bool PGSIMeta::PgsiMeta::operator==(
                    const PgsiMeta & obj)const {
  return dataList == obj.dataList; }
```

That is using a fully elaborated name. My preference is definitely for the former.

The Winner of SCC 18

The editor's choice is Simon Seabright. Please email `francis.glassborow@ntlworld.com` to arrange for your prize.

Student Code Critique 19

I'm trying to write a class to represent a card that can used to create a pack of cards. I'm thinking an array of pointer to card in the back of my head.

```
class Card {
  public:
    Card():itsNumber(0) {}
    Card(int Number):itsNumber(Number) {}
    virtual ~Card(){};
    void SetNumber(int val) {itsNumber = val;}
    int GetNumber() const {return itsNumber;}
    virtual void Display() const =0;
  private:
    int itsNumber;
};
const int RegularPack = 54;
int main() {
  int i;
  Card* pack[Regularpack];
  Card* pCard;
  for (i = 0; i < RegularDeck; i++) {
    Card:pack[i]->SetNumber(i);
  }
  cout << pack[50]->GetNumber() << "\n";
  for (int i = 0; i <= 53; i++) {
    cout << i << endl;;
    cin >> *pack[i]->SetNumber(i);
  }
  cout << "The CARD is: " << Card << endl;
}
```

This code contains a variety of different errors. First there are several design flaws that need addressing. Then there are syntax errors, at least one of which suggests that this is not the writer's first programming language. And finally there are issues with consistency of idiom. A successful critique should cover all these issues but prioritise them in an appropriate manner. Style, correctness, completeness are all elements that are taken into account when assessing a winner.

Francis' Scribbles

Francis Glassborow

ACCU & Programming Languages

Back in 1987 when this organisation was founded it was called 'The C Users Group (UK)'. It was started, as were most user groups at that time, by a small band of enthusiasts. Membership was £10 and lasted for six issues of C Vu. At that time C Vu was a typical collection of A4 sheets stapled together. Much of the content was reproduced from elsewhere. While I notice, I think that the long-term management of ACCU deserves a great deal of credit for holding membership fees down. £10 in 1987 was worth somewhat more than the current basic ACCU membership fees, yet today you get much more for your money than those enthusiasts did fifteen years ago.

By the early 1990's CUG (UK) was neither limited to C nor to the UK. C++ was so clearly of interest to most C programmers that most of us thought it perfectly reasonable to add it into the range of articles published in C Vu.

Several things then happened that have been documented elsewhere with the result that CUG(UK) changed its public name to 'The Association of C and C++ Users.' Of course this got abbreviated to the acronym ACCU.

As the organisation slowly grew during the 1990s the material being published in our two magazines (and for a time a newsletter for Acorn programmers) broadened. We published what members wanted to write about. That is a sound basis for editorial selection, if a member wanted to write about it, probably others would be interested in reading about it.

Objective C never really made it to our pages, not that there was any reason that it shouldn't, just that no one was interested enough to contribute anything. Java hove on the scene and now there was clearly something that many members would be interested in. At this stage it was clear that there would be other languages that a substantial part of the membership would be interested in. The ACCU Committee decided that constant name changes would destroy the brand image and that we would be best off by branding ourselves with the ACCU acronym. In doing that we followed many other successful groups. Who now knows what ECMA originally stood for? Or ACM? Or ... (don't write and tell me, because I do know the answers). These days what the acronyms stood for has been discarded. In the same way ACCU is now ACCU and has no intention of following all those organisations with a death wish who want to rebrand themselves.

So now let me turn my attention to Python. What makes Python different from awk, Perl, Ruby etc? I do not know that I can put it into words but clearly it has something that is attractive to many highly competent C++ programmers. When Andy Koenig noticed that the ACCU Spring Conference 2002 was hosting the UK Python conference he wrote to congratulate me (as Conference Programme Organiser). By that time Andy was not only regularly programming in Python but had also spoken at a major Python conference in Japan.

Then I noticed Boost was actively working on a library to provide bindings in C++ for Python. Python had already provided mechanisms for the reverse.

Can these highly competent C++ programmers be confused or does this braceless language belong in the community that ACCU represents?

I know that the awk, Perl, etc. enthusiasts and plain simple users wonder why Python seems to be so actively coming into the broad ACCU community. The answer is plain and simple, because they feel at home with us and want to participate in ways that some other language groups do not.

From my perspective they bring a welcome breath of fresh air and pose questions that we will benefit in finding answers to. Someone who is fluent in both C++ and Python is going to be a much more rounded programmer than one who only knows C++.

I know that some members fear that Python will somehow usurp resources that they feel should go to the older more traditional languages. I think they are wrong. As long as members write about C++, ACCU editors will publish articles on C++.

However there is another issue with this broadening of ACCU's base. The C# standards (yes it is a standardised language and runtime system) were issued by ECMA and are in the process of being fast tracked to ISO. The UK language experts were concerned about leaving maintenance entirely in the hands of ECMA because most of us are individuals and not backed by large corporations. The way ECMA works means that most contributions have to come from fully paid up corporate members. There is no equivalent to the National Body participating that is the way ISO works.

However there is one option that ECMA provides and that is for purely voluntary organisations to be allowed to participate in ECMA workgroups without charge.

It both amazed and delighted me to learn that one proposal to allow independent language experts to participate in the future standardisation of C# was to accredit ACCU as participants in the relevant workgroups.

I have no idea if this will actually come to fruition but I think that the whole membership of ACCU (past and present) should feel very happy that in fifteen short years we have moved from a bunch of young enthusiasts for C to become an international organisation with sufficient status to be considered for such a responsible task. For me, ACCU has at last grown up (you would have to read some of my editorials to understand that) and become a broad community of people who consider programming as more than just the language in which they write source code.

Yes, please do write and express your own views but when you do so remember that one of the great qualities of ACCU members is that they are not language bigots. If enough members wanted to write and read about Ruby, I am sure we could find the space for them just as we always find space for those that write about C (I am sure your editor feels the same way as I do, I wish there were more of them).

In so far as ACCU conferences are concerned, I think you have little reason to worry that speakers on other languages might dilute them. Rather I think they have been and will be enriched by such speakers. Even the most dedicated C++ or C programmer should realise that we have things to learn from experts on other programming languages.

Future C

Some of you may know that there has been work going on over the last three years concerned with providing support for programming DSPs (Digital Signal Processors) in C. This support constitutes a major part of a TR (Technical Report) that is being worked on by WG14.

It is quite clear to me that the sub-community of C programmers whose work lies mainly in the domain of DSPs need the proposed extensions to C, or at least something like them. They need a Standard for exactly the reason that computer languages have them; to provide portability both between hardware and between compilers.

Unfortunately DSPs have some heavy requirements. They need a highly specialised type of fixed-point arithmetic, not one that would be of any use in such sectors as the financial one. They also need to be able to control the behaviour with overflow. Effectively they need twelve new fundamental types and each must support three types of overflow management. The current proposal is to provide a new type qualifier so that programmers can either use the default behaviour of the new fundamental types or qualify them to provide saturated arithmetic.

The proposal to have a second type qualifier to allow for modulus wrapping behaviour has been withdrawn and either a #pragma or a set of functions will provide that flavour. Even so we have 24 new types and a bundle of rules to deal with conversions between them and between them and the existing arithmetic types.

I think most C programmers will wish to resist any attempt to add those to Standard C. Currently we are only looking at a TR, but it is clear from my discussions with those working on this TR that they will want incorporation of support for DSPs into the next release of C.

Is there a way round which will satisfy a large majority and that the rest can tolerate? I think so, but it will need a change of view.

Some of you may know that the most recent release of C added support for extended arithmetic including adding complex number types and providing some genericity for maths functions, but only those in the standard math library.

Quite a few programmers think that was unwise because it made C larger and more complicated though the new features were only useful to a limited part of the C community. The antecedents of those new features can be found in a technical report released in the mid 1990s. In other words we have already been down this path and not everyone is happy with the result.

It seems clear to me that we need a highly stable kernel C in which new features are only added if they demonstrably strengthen C for all the community that use it. That should include a major part of the C++ community who need compatibility between the C they use and C++.

I would advocate a radical review of the current version of C with a view to removing a number of features, including but not limited to the support for complex numbers.

Now I can feel a sense of deep disquiet among many who have legitimate needs for those features. But before they ignore this proposal, please read on and see the rest.

There will always be sub-communities who have a need for other features that are supported across compilers and across hardware. The

numerical experts have a cluster of things they need to have supported. The financial world has its desirable features. Those involved in embedded programming have clear needs (for example some standard support for bit variables would, I think, be helpful to those working on 8051 derivatives). And, of course, the DSP programmers have all those needs that are being addressed in the current TR.

Note that these needs cannot easily be met by something entirely separate (like the Posix Standard) because each sub-community needs specific additions to fundamental C. They need new types, new type qualifiers etc.

What I am suggesting that we consider is producing a highly stable C Standard that will only be maintained in the future. And then work on separate standards for extensions. It should then be possible for a compiler vendor to market C, or C with financial extensions or C with DSP support etc.

Note that this is not sub-setting the C language, though you might view it as super-setting. There is a long history of bad experiences with sub-setting computer languages.

I do not think I am just playing word games. It does matter whether you start with a whole and cut bits out of it or start with a base and allow additions to it.

C++ programmers should feel relatively easy with the idea because it is only the language equivalent of derivation. If you need Standard C any of the extended versions will do (the principle of substitutability) but if you have C with DSP support, it will not help if you need C with numerical extensions.

If we pursue this path and work carefully on the exact specification for the pure C kernel we might even be able to put the issue of compatibility between C and C++ to final rest. C++ would become one more extension to C, albeit a very large one.

Extended Operators

Another issue that came out of discussions about the type system and DSP support was the issue of extended operator behaviour. All that the saturated type qualifier does is to change the semantics of overflow. Yet to achieve this it makes major changes to the built in types of C. That seems overkill to me. And I am not alone in this.

Currently we have only two other options, we can abandon using operators and do everything through function calls. But the frailty of this approach is one of the main motivations for C++ providing operator overloading. Or we can use standard #pragmas to change evaluation behaviour. That seems fraught with potential for errors.

I have an alternative, which is to allow operators to carry modifiers. We discussed this as a theoretical idea at the recent WG14 meeting. We could not go further because the idea raises serious compatibility issues for C++.

For what it is worth the idea is that C would support a syntax so that I could write, for example:

```
a = b +[sat] c;
```

Note that this would allow the qualification to go where it belongs, on the evaluation and not on the type. That sidesteps the problems with rapidly growing lists of type conversions.

I think the idea was popular enough so that we might have added it to the main part of the TR had there not been a C++ issue. As it is, it will go as a suggestion in an appendix.

So what about the C++ issue. C++ already allows overloading of operators. It would certainly be possible to allow such overloads to carry an extra parameter that could be used as an extended qualifier. In a way we already do that with `new(nothrow)`.

I can imagine that such an extension would be helpful in some application domains. For example, vector mathematics uses two products – cross and dot. Currently we have to choose which of those we will represent with * and then what other operator we will pervert by using it as the other product. This adds a burden to the domain expert who has to remember which is which.

Would it not be better to be able to write:

```
a = b *[dot] c *[cross] e;
```

Not least because the domain expert would immediately recognise if that is well formed.

By the way, the syntax has been carefully checked by those who are familiar with syntax problems. The recycling of the index operator was chosen for two reasons.

First the way the index operator is used already post-qualifies an expression so we are only extending it to post-qualify an operator.

Second, and probably more important, as far as we could determine that syntax would not cause the kind of parsing problems that C++'s use of angle brackets for templates has caused.

Feedback

As always, I would be very happy to have feedback on the ideas that have been presented in this column. They are pretty raw and are largely presented here to get other people thinking, and possibly coming up with better solutions.

There are real problems that need tackling and the more minds that can consider them the better chance we have of getting good solutions.

Problem 5

Consider the following brief program:

```
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;
}
```

What is the output and why?

I had a couple of responses to this from people who were puzzled as to what the problem might be. Let me ask you a simple question, what is the type of `*x`? Well what is the type of `x`?

The answer as we all know is that it is a pointer to base. That means that `*x` must be either a base or a reference to base. Does it matter? Well, in some circumstances it might not but in this case whatever it is must be copied because that is the requirement for exception objects. There are no such things as virtual constructors so the type of the thing copied will have to be the static type of the dereferenced pointer. In other words the process of throwing converts the derived object into a base one.

The catch clause will catch a reference to a base that actually refers to a base. That means that if your compiler behaves correctly the message should be:

```
base
```

The other points that you should have noted is that:

1 The dynamic instance of a `derived` is never deleted
2 Should we be using `std::endl` or should we prefer `'\n'`?

That second question is one that is often skimmed over. `std::endl` does two things, it appends a newline character to the output buffer and then it flushes it. As `cin` and `cout` start out in life bound together, we do not normally need to force a flush. When input is required from `cin` the buffer attached to `cout` will be flushed.

Problem 6

Consider the following definition of a simple function to draw a line across the screen. Assume that the resolution has been chosen so that successive pixels can be generated by incrementing an `int`.

```
void yline(int y, int startx, int endx) {
  for(int x = startx; x != endx; ++x)
    plot(x, y);
}
```

There is a flawed assumption in this code. How should it be fixed? There is also an issue of the line itself. Where does this line end? Is that a reasonable place for it to do so? In the computer world where pixels have finite size, how should we represent a line of zero length?

Another way to represent a line is by providing its start point and its length. What problems might result from such a choice?

Features

Professionalism in Programming #17

The Code That Jack Built

by Pete Goodliffe
<pete@cthree.org>

Unless you've started reading this article by some unfortunate accident, you are a programmer. You write code. That is, you're usually found hunched in front of the ethereal glow of a monitor, entering profound combinations of punctuation characters into some form of text editor.

If that was all there was to developing code our job would be a great deal easier, although we'd run the risk of being replaced by a proverbial infinite number of monkeys with their infinite number of text editors. Instead, once we've written our source code we run it through a compiler (or interpreter) to get out something that actually functions (or might just function) as we intend. Rinse and repeat.

The task of converting our carefully honed high-level language into an executable that can be distributed is commonly referred to as "building" code (although you'll find that this term is used pretty interchangeably with "making" and "compiling" in most contexts). Building is often used as a metaphor for programming. See the sidebar for some thoughts on this comparison.

This act of building is a fundamental part of what we do – we can hardly develop code without performing builds. It's important, then, to understand what's involved and how our particular project's build system works in order to have confidence in what we're up to. There are a lot of subtle issues at play here, especially when a code base gets to a reasonable size. Interestingly, almost all programming textbooks will gloss over this kind of topic, they present single file example programs not showing any real build complexity.

For some developers the build system provided by their IDE is sufficient, but this doesn't remove the burden of understanding how it's working. It's very convenient to hit a button and have all your code generated, but if you don't know what options are being passed to the C compiler, or what level of instrumentation is left in your object files then you're not really in control. The same holds if you're firing off a single "build" instruction at a command prompt. You must understand what's going on under the covers to be able to repeatably perform reliable builds.

Making mountains out of molehills

Let's spend a little time investigating what all this fuss is about anyway.

Say you're starting a new project, coded in C. It will solve all the ills in the software development world, and will usher in a new era of world peace. However you have to start somewhere, and all you have at first is a single file containing `main`.

It's easy to build and run this single file program, you just type "`compiler main.c`"[1] and out spits an executable for you to run and test. Simple.

The program grows and to help organise the parts you split it out into multiple files, one per functional block. The build is still a simple process. Now you type "`compiler main.c func1.c func2.c`". The same executable program is spat out, leaving you to carry on testing as before. No sweat.

Soon you recognise that some sections of the code are really individual components with particular concerns, almost like stand-alone libraries. It would be easier to reason about these sections of code by placing them all in their own directories – grouping the like sections of code together. Now the project is beginning to spread out. The simple way of building under this new file structure is to just build the source files in each individual directory by hand, firing off a compiler call that doesn't build an executable, but just intermediate 'object' files. This is done in each of the directories, then the `main.c` file is compiled, and linked in with all the other intermediate object files. To do this, you'll also have to help your compiler out by telling it where to find some of the other directories' include files as required. Hmm, things are now getting a little more complex.

Whenever you change some code in one of the directories you have to fire off the compile command in that directory, and then issue the final link command once more. Quite manual. Additionally, if you make a change to a function declaration in a particular header file, and there are other directories with code that depend on that header, those directories have to be rebuilt. You'll usually know you didn't recompile these dependent files because your final link stage will generate cryptic complaints.

To fix this huge command line burden you can write a shell script (or batch file) that walks around each directory and fires off the requisite build commands. Having hidden all that messy work and the tedious compiler parameters, you can get back to the serious business of code development, safe that you don't have to memorise unnecessary fluff.

Now it occurs to you that a number of these subdirectories of code really *are* stand-alone libraries. They can also be used in other projects, not just this particular one. You tidy up the code so it's a little friendlier to use, add some good user-facing documentation, and then alter the build commands so that they generate shared libraries rather than object files. This requires some more changes to your build script, but it's relatively hidden so it's not too painful.

1 Obviously, replace compiler with the command to prod your C compiler – this is a hypothetical example.

Do we really "build" software?

We frequently think about the software development process as being like "building". It would indeed be hard to argue that we aren't building software. This metaphor naturally equates what we do with the "traditional" building industry. We have, in fact, seen some sort of overlap and collaboration between these two disciplines, as the patterns movement learns from Christopher Alexander's work [1].

It's valuable to understand how far this metaphor works and is useful. No metaphor is perfect, after all. Although it is philosophy and a bit of an aside, it's still worth spending a little thought power on since the comparison will inevitably prejudice our approach to development.

Like the physical construction process of, say, a house, we start from nothing and build by placing layers of structure atop each other. Before the construction begins, a process of requirements gathering and careful design/architecting should have been performed. Whilst you can probably build a garden shed without much planning, you'd be daft to hope an unplanned skyscraper stood a chance of standing up; you'd need some serious design and planning up-front. This parallels our software construction neatly.

The metaphor stretches thinly in other areas though. We can modify the foundational layers of our software constructions far more easily (although there is still danger involved), and it's much cheaper to tear down a software edifice than a physical one. This means that in the software world we have the opportunity to prototype and explore more than in the physical world. Real world building mandates sound engineering principles, this is enshrined in statute. Many software firms wouldn't know an engineering principle if it slapped them in the face.

What we're thinking about in this article revolves around the *procedure* involved in this building task. The metaphor's a bit out of kilter here too, at least for the purposes of this article. Our entire development procedure *is* akin to a physical construction process, comprising system conception, design, implementation, and test. But at a lower level, each time we take a fresh copy of some source code we perform a "build" on it, and that's what we're looking at here.

Development carries on like this for some time. Code is added rapidly. Many new sub-directories and sub-sub-directories get created. Although the file structure seems pretty neat, build times become a problem since each time you fire up your build script it goes around recompiling every source file, even if it hasn't changed and it doesn't depend on a header file that's changed. The temptation here is to fire sub-directory builds off by hand again and track the changes yourself (perhaps by running individual directory shell scripts as a half-way-house). The project is now so large it would be very easy to miss some dependencies and have hard to resolve build errors, or worse, issues that don't stop the link working, but that do make the program behave in incorrect ways.

Now your development is on the brink. You can't trust the system being used to build the code. It's not safe. You can only really trust the executable if you've done a complete clean out and rebuild from scratch.

Enter the tool for just this occasion. The classic solution is a command line program imaginatively called ***make***. It deals with all of your intermediate object files and compilation rules for you, and most importantly tracks which files depend on which other files. You tell it what to do by writing *makefiles*, which provide the necessary build rules. It looks at the source file timestamps to check which files have changed since you last performed a make, and then recompiles those, and any other files that depend on them. It's a more intelligent version of the shell scripts above, specifically tailored to the task of compiling and recompiling software. A deeper description of how to use the make tool is outside the scope of this article, but it's something every developer ought to know well. Over the years many variants of humble *make* have appeared, these days many with GUI façades.

Building builds

In that sinking morass of software construction we've see some of the issues of a 'build' procedure. Essentially, any build process takes source files as input, and out the other end it spits some executable program, or even perhaps an entire 'distribution' (including help files, installer etc, all packaged neatly and ready to be burnt onto CD).

Like the cumulative story that I shamelessly pilfered this article's title from, as our software develops and matures the build process develops and matures with it. Maybe yours didn't start in quite such a basic state as in the example above, but generally it's true: the makefiles start off simple at first and grow alongside the code they build. A large project often has a bewildering build process that requires (but doesn't necessarily always have) adequate documentation. We can see that the act of compiling a single source file is at the lowest level of the build food chain, we raise a tower of extra work on top of this simple act.

It's worth bearing in mind that a build process is not always just about compiling C/C++/Java source files. It may also involve preparing some text registration files from templates, creating internationalised strings, or converting graphics files from their source format to some destination binary format. Practically all such activities can also hang off the standard makefile system, and be run in the normal course of a build. This does presume that the tools involved are "scriptable", and can be run in a command line environment.

What makes a good build system?

There is a general set of considerations for a good build system. I'll spell them out below. Maybe you're creating a new build system from scratch, or you're improving, or even just evaluating your existing system against this yardstick. As these are only a broad set of rules, they exist to be broken; they can't possibly apply to all scenarios.

Mechanics

The ideal is that a build system should take undoctored "virgin source" and compile it all in one go, with no human intervention. There should be no "special steps" you have to go through to perform the build. You should not have to fire up some GUI app half way through and prod a file. You shouldn't even need to run more than one command to perform the build stage. This ensures that no information is locked away in your head, just waiting to be lost. All the build magic is 'documented' in a reliable place. The build is always repeatable. It's safe. (More on safety below.)

If you can't reach this ideal (and it's not at all unreasonable), then the less manual a build is, the better. All the manual steps need full documentation. It's usually acceptable for one step to obtain the virgin source, one step to build it, and one to create the distribution from this, for example.

If your build procedure is a simple matter of firing off one command, you can easily set up overnight builds of the entire source tree. This is a remarkably helpful trick to identify build errors early on. When you add an automated regression test suite into this mix you have a potent tool to validate your work on an ongoing basis.

Targets

A makefile allows you to specify multiple *targets* for a build. These allow you to use the one build system to generate several different outputs. The targets may be something like these

- Different platforms to build your application for (PC/Apple/Linux, Desktop/Embedded)
- Product variants (full release or demo release with save/print disabled)
- Development build (with debugging support enabled/disabled)
- Differing "levels" of build (build just the internal libraries, build the application, build entire distribution)

Each of these targets can be simply built from the one source tree. There is a huge benefit doing this rather than having separate source trees for each target. If most of the code is the same it would be an intense task to keep several similar trees in sync excepting their minor differences. The physical differences between these targets that the build system will accommodate might boil down to a number of things:

- Different files being built (e.g. `save_full.c` or `save_demo.c`)
- Different macro definitions being passed through the compiler (e.g., compiler flags altered to predefine `DEMO_VERSION` when building all source files, to select `#ifdef`ed code)
- Different toolsets/environment being used for building (e.g., for different target platforms)

For every build target rule in the build system, there should be a corresponding 'clean' rule, that undoes all the build operations, removing the program executable, intermediate library and object files, and any other files created during the build. The source tree should revert to its original 'virgin' state; it's relatively easy to verify. This implies that build systems that physically alter any original files are nasty. You should instead use the original files as templates, and send modifications to a different output file.

Whilst you could have any number of targets for all sorts of minor differences, it opens the possibility of making your build system complex and unwieldy. Some of this configuration can be done at code install time, or even run time. This would be preferable if it reduces the number of different builds that exist and require testing.

Reliability

Builds should be deterministic and reliable. You should be able to determine the set of input files easily before performing the build. You should then be able to take these files and perform a build step, followed by the clean step, and get back the same set of files you started with. Performing the build again should give you the same final executable you got the first time – the build should be repeatable. You should be able to mark (or archive) a set of files as a particular version of the software, and then be able to perform many identical builds of it.

A build process that spits out an unreproduceable binary is worrying. If what comes out of a build depends on which way the wind is blowing, the world becomes a hard place to reason about. This means that gratuitous use of `__DATE__` or other potentially changeable information should be kept to an *absolute* minimum in a build tree.

Safety

Our build systems should be safe. It sounds good, but what does that actually mean? We've already said that our builds should be reliable and repeatable, but there's even more to it than that.

First, the system should be documented thoroughly. A newcomer should be able to immediately see how to build the code without spending years studying messy makefiles.

You must be able to pull out some source from three years ago and rebuild it correctly. This is crucial, since an important customer may find a significant bug in an old revision of software, and if you can't get back to that version and generate the exact same program you may never be able to reproduce, let alone find the fault.

This implies that all your source should be appropriately accessible to feed into your build process. There is no excuse not to keep the source code under source control (e.g. CVS). Only the 'input' files that are part of the virgin build tree need to be under control. No object files, libraries or other generated intermediate files should be held under source control. Obviously, the makefiles themselves will be held in source control too.

Safety also requires that the build environment should be reproducible. To make this practical there should be no unreasonable dependency on type

of computer installation. There should be no ludicrous reliance on setup in environment variables. That said, it can be hard to determine the exact environment you're building under to be able to reproduce it later. The build tools and operating system configuration you're using should be accessible at any future point. The compiler doesn't necessarily need to be held under source control, but you need to document exactly which version was used, including patches/service packs etc. This kind of issue raises some ugly problems; how dependent on hardware (i.e., a particular PC) is your build environment, and do the libraries, service packs, etc on the build machine affect the produced executable at all? For example, MSVC *should* produce identical code on Windows 98 and Windows XP, gcc version 3.2 should produce identical code under Red Hat 7.3, SuSE 8.1 and Solaris.

Builds should not be noisy. If your code elicits compiler warnings then there is something there you should be looking in to. Persuade the compiler to be quiet. Copious compile warnings can cloak any more insidious messages that you should be reading. You should really be building with all compiler warnings enabled for maximum peace of mind, switching them off does not fix the problem.

Most importantly, your build system must not carry on after an error. It should stop and leave you in no doubt what broke and where it can be fixed. If the build process continues, other problems will almost certainly result as a consequence of that first skipped error that will be very hard to understand. For your own sanity, don't break this rule!

To be a safe and resilient build system it's best not to be too clever. Complex build tools that do things you don't understand are a worry[2]. The makefiles you create should be simple and tidy. The temptation is always to quickly hack up the build system and spend more time on the code being built. However, it must be easy to modify the makefiles when the project requirements change. Anything that requires a guru build engineer to update is a liability.

Authority

Who should be able to perform a build? There are two different answers to this question. We'll look at the first here, and the second in the next section.

The build system should not only be comprehensible to a build guru. Quite simply, any developer should be able to perform a build. In fact, every developer *must* be able to perform a build or they wouldn't be able to develop. To be even more accurate, every developer must be able to perform a *development* build. For release builds, see below.

This again means that the build should be simple to learn (and well documented). It should be easy to modify and maintain. Each developer will be working on their small part of the overall system. For this reason it makes sense that there should be targets to build each small part of the system, so there is no need to rebuild everything all the time. This is often achieved using a 'recursive' make technique. The same makefile template is copied into each directory (perhaps it includes a global one for the common rules). This makefile recurses into each subdirectory, builds the components in that directory, and returns. In this way you can type "make" at the top level directory, or just in the directory for a particular component, and what you want built gets built.

Please release me

Some builds are particularly special, and require more care in their preparation. These are *release builds*, builds that are made with special purpose, rather than just in the course of normal development. A 'release' could be one of a number of exciting events, a beta version, the first official product release, or a maintenance release. It may also be an internal milestone release, or an interim release to the test department; these won't leave the company, but are held in as high a regard as other release builds, almost a 'fire drill' for an official release build.

Now if our build system is carefully crafted, there shouldn't be too much extra preparation needed. However, these builds must be handled thoughtfully, we need to make sure any build issues don't compromise the code.

Such important builds should always come from a virgin source tree, not from someone's half-built working tree. Before the code build is performed, a specific step beforehand identifies which source code, and which particular version of each file to include in this release, and then marks it in some manner, usually by tagging or labelling it in the source control system. The release files are now obtainable at any point thereafter.

Each release build has a particular name you identify it by, sometimes a cool codename, sometimes just a build number. This will usually tally with the source control label the code was marked with. If you and I agree we're talking about "build five" when investigating a fault, then we're both singing from the same hymn sheet. If you're working with build five, when I found the fault in build six how do we know we'll see the same issues?

There may be some extra packaging stage after the code has been built, i.e., prepare a CD, add documentation, integrate licensing information, or whatever. This might also be automated.

After being built, each release needs to be handled specifically. It should be archived and stored for future reference, presumably these days on a CD. Obviously you store a copy of the final built executable in whatever form it ships to the user (the exact shipped zipfile, self-extracting exe or whatever). You should also capture the final state of the build tree if possible, but in most projects this may be enormous so it might not be practical. At the very least the *build log* should be retained, that is that exact sequence of commands issued and the response generated, so this must be capturable and captured. These logs allow you to look back over old builds and see what compiler errors may have been overlooked, or exactly what happened during the build. Sometimes, this gives a clue into a fault reported in a years-old version of a product that has long since been discontinued.

Each release has a *release note* that describes what changed in this release. It may or may not be a customer facing document, depending on exactly what you're building. These should also be archived. The release note contains updates subsequent to the printing of the official documentation, any known issues, etc. They are an important part of the release procedure and shouldn't be overlooked.

When performing release builds you must select the correct set of compiler switches – they might differ from those used in development builds. Any debugging support gets switched off, for example. You also need to choose what level of code optimisation should be selected[3]. The optimisation level may be lower for development builds since the compiler's optimisation stage often takes a particularly long time to execute. This can become unbearable for very large build trees. However, if you use different sets of compiler options for development and release builds, beware. You *must* be testing the exact builds you release regularly. Aim to minimise the differences between release and development builds.

Since creating a release build is a relatively involved task, and important to get right, it is usually given to one team member (perhaps one of the coders, perhaps someone in QA). That person produces all the builds for that particular project, to make sure that each build is of the same quality. Release builds are as much about procedure as they are about the build system.

Jack of all trades, buildmaster of?

Many organisations employ a specific person to fulfil a build engineer role, often known as the *buildmaster*. The role may also involve planning and managing release schedules, or it may be purely technical. The buildmaster will be the person who knows the build system intimately. They've probably set it up at first, they add new targets as required, they maintain the overnight build scripts, etc. They also own the build system documentation.

They are tasked with performing the release builds, and for this reason are often heavily involved with the configuration management system being used. They are charged with ensuring the reliability and safety of the release process.

The buildmaster is not always a distinct engineer, sometimes a coder doubles in this task. However, it is helpful to have just the one nominated person owning the build and release aspects of a project. This is not an excuse for 'normal' engineers not to understand the build process, though.

Conclusion

On the face of it building software is easy, if you have the right tools. But you do have to know how to use the tools properly. Producing trustworthy builds for production is a more involved matter. It is important to have an understanding of what's going on when you fire off a build, even if you don't have to alter the build system every day.

Performing good builds is not a completely straightforward task; our jobs are safe from the proverbial infinite number of monkeys. They're too busy arguing about which of their infinite number of text editors is the better one, anyway.

Pete Goodliffe

References

[1] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, 1979. ISBN 0 195024028.

2 I have an in-built distrust of anything more clever than GNU make, but that probably says more about me than the other clever make tools. GNU make is quite clever enough, thank you!

3 This can be a harder question to answer than you think. Ramping the optimiser up to warp speed nine may expose compiler optimisation bugs that break your code.

Using SAX Parsers

Tim Pushman <tpushman@gnomedia.com>

This article will introduce the subject of parsing XML files, using as examples the Expat parser and the Xerces parser. In the process we will examine the two event interfaces for XML parsers, SAX1 and SAX2. I will assume that you've read the two previous articles in the series ([1] and [2]) and I assume that you have a good understanding of C++. The article won't cover the design of XML documents, the samples we use will from necessity be simple and designed to demonstrate the basic facilities of the XML parsers. We will create a simple program to parse an XML file and count the characters and tags in it, showing how the program differs between Expat and Xerces.

I'm assuming that the intended audience has never used an XML parser before, if you have you may want to wait till further articles appear. My intention is to give a basic overview of setting up a parser and reading in an XML file.

The two interfaces we will play with, SAX1 and SAX2, are called Event Based APIs and are straight-forward interfaces utilising callback functions. Originally designed for Java, they are available in many other languages, including C and C++. The original SAX1 API was designed by the members of the `xml-dev` mailing list in 1998 and released as a 'de facto' standard to the programming community. In May 2000 SAX2 was released which included the use of namespaces, filter chains and methods for querying and setting parser properties and is the recommended API to use for current applications.

Strictly speaking the SAX API is designed for Java and is described by a set of Java classes . Although the API has been ported to other languages (such as C) the ports do not, and cannot, mirror the Java API exactly. I use the term SAX rather loosely in this article to describe event based XML parsing.

In particular, Expat is a C based parser and has no classes as such, so the interface is of necessity an approximation. Expat is also based on the SAX1 API, so you may wonder why we are going to start the article with looking at how to use Expat. Well, for one thing, Expat is one of the most widely used XML parsers in the C world (it's also the basis for the Perl and PHP XML modules). It is extremely fast and has a small disk and memory footprint. If you want to use XML in your own applications, Expat may be the first thing that you look at. And finally, there are some C++ wrappers available, one of the best being Jez Higgins' SAX in C++ [`http://www.jezuk.co.uk/`].

The Interfaces

As I said above, the SAX interfaces were originally designed for Java programs, they exist in the `org.xml.sax` set of packages and consist of interafces, classes and sub-packages.

In SAX1 there are interfaces for the parser, handlers, exceptions and so on, SAX2 kept the same basic structure but deprecated some of the interfaces and added some new ones. For example, `Parser` is now deprecated and replaced by `XMLReader`. There are often also a set of helper implementations that provide barebones functionality, such as the `SAX2XMLReader` implementation of the `XMLReader` interface.

I'll describe some of the main classes in SAX2 here using the Java names, how that works out in the C Expat parser and the C++ Xerces parser will become clear later in the article. I'll mention when the class is replacing one of the SAX1 classes

The first set of classes are those used for parsing an XML file. They are based on the `XMLReader` interface (`SAX1: Parser`), and usually created via an `XMLReaderFactory`. Most implementations provide an adapter implementation called `SAX2XMLReader`. The `XMLReader` interface provides methods for setting and getting features and properties for the parser, setting handlers and a method called `parse()`. Typically we derive a class from `XMLReader`, create it, set the properties and features that we want, set the handlers and then call the parse method. Parsing can throw exceptions, generate errors and warnings and call the handlers.

The handlers are used for handling the various events and come in different flavours. The main handler is based on the `ContentHandler` (`SAX1: DocumentHandler`) interface. There are also the `ErrorHandler` (errors, warnings and fatals), the `DTDHandler` (DTDs) and the `EntityResolver` (for external entities). There is also a `Locator` class that is used to keep track of where in an XML file the parser is.

The `XMLReader` is normally passed an `InputSource` (which encapsulates an input stream) and during parsing can throw various `SAXExceptions`.

There are various helper classes and interfaces available, such as the `Attributes` class to encapsulate attributes, different adapter and implementation helper classes and a factory class to produce the required parser.

In a nutshell, to parse an XML file, you must create a parser, tell it what functions will handle the various events it creates from the file and then let it rip on the file.

The Xerces parser has pretty much the same classes and structure as the Java SAX API (along with other classes for DOM and so on), but Expat, because it is written in C, has a set of functions that try to mimic the above functionality as much as possible. Next we'll have a look at Expat.

An Example XML Document

We need a simple XML document to play with. I could start off with an example using the hypothetical Person who is part of a hypothetical AddressBook, but I won't. I've seen enough of them and I refuse to write another... instead we'll look at a hypothetical configuration file for a hypothetical application.

Let's assume that we have an application that stores its state (user name, last used file, etc.) at shut down in an XML file. And reads this file at start up to restore its state to what it was before.

Listing 1 shows the first cut at this file, we'll expand on it further as we go.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE config>
<config datecreated="20011210">
  <user>John Smith</user>
  <login>jsmith</login>
  <password>topsecret</password>
  <lastfiles>
    <lastfile timestamp="20011210T1002">
       accounts.txt
    </lastfile>
    <lastfile timestamp="20011190T1132">
       /home/jsmith/docs/letter.doc
    </lastfile>
  </lastfiles>
</config>
```

Listing 1: The sample XML file, version 1

Using Expat

Expat was originally written by James Clark, one of the pioneers in the world of XML and SGML. It's written in C and is available on many platforms. Recently development of the parser moved to SourceForge, where it is overseen by Clark Cooper. To compile the following examples you will need to download and install the Expat parser from `expat.sourceforge.net` (I used the 1.95.2 version for this article).

Listing 2 (next page) shows a bare bones program that creates a parser, parses a file and exits. In the process it counts the number of characters, tags and attributes that it reads.

On its own, not a terribly exciting application, but it demonstrates the four fundamental actions in using Expat to parse an XML file: create a parser, assign event handlers, give it data to parse, and then free the parser.

All the Expat functions are prefixed by "`XML_`" and they all take an instance of the parser as their first parameter (except, obviously, the `XML_Create function`).

We create a parser using `XML_Create(NULL)` which returns a pointer to an `XML_Parser`, and it is this pointer that we pass around to the other functions and is finally used in `XML_ParserFree(p)` to free the parser.

`XML_Create` has one optional parameter, the document encoding, which overrides the document encoding specified in the document itself. In this case we pass `NULL` to indicate that the document will specify the encoding, or else to use the default UTF-8 encoding.

The interesting stuff happens between these two calls to create and free the parser. As the parser reads through the file it will generate events

```
#include <iostream>
#include <fstream>
#include <string>
#include <expat.h>

#define BUFFSIZE 2048

typedef struct {
  unsigned int tagCount;
  unsigned int attrCount;
  unsigned int charCount;
} ElementCounts;

static void startFn(void* data, const char* el,
                    const char** attr) {
  ElementCounts* pc = (ElementCounts*)data;
  pc->tagCount++;
  for(int i = 0; attr[i] != NULL; i += 2) {
    pc->attrCount++;
  }
}

static void endFn(void* data, const char* el) {}

static void characterFn(void* data,
                const XML_Char* ch, int len) {
  ((ElementCounts*)data)->charCount += len;
}

int main(int argc, char** argv) {
  if(argc < 2) {
    std::cout << "Usage: test01 some.xml"
              << std::endl;
    exit(1);
  }
  std::string filename(argv[1]);
  std::cout << "Using " << filename.c_str()
            << std::endl;
  std::ifstream ifs(filename.c_str());
  if(ifs.fail()) {
    std::cout << "Error opening input file,
                  exiting..." << std::endl;
    exit(2);
  }
  XML_Parser p = XML_ParserCreate(NULL);
  if(!p) {
    std::cerr << "Failed to create parser"
              << std::endl;
    exit(3);
  }
  ElementCounts* pec = new ElementCounts();
  pec->tagCount = pec->attrCount = 0;
  XML_SetUserData(p, pec);
  XML_SetElementHandler(p, startFn, endFn);
  XML_SetCharacterDataHandler(p, characterFn);
  // parser ready and raring to go.
  bool done = false;
  int len = 0;
  int totalCount = len;
  char buff[BUFFSIZE];
  while(!done) {
    ifs.read(buff, BUFFSIZE);
    done = ((len = ifs.gcount()) < BUFFSIZE);
    totalCount += len;
    if(ifs.bad()) {
      std::cerr << "Error in read operation."
                << std::endl;
      exit(4);
    }
    if(!XML_Parse(p, buff, len, done)) {
      std::cerr << "Parse error at line "
                << XML_GetCurrentLineNumber(p);
      std::cerr << " with " << XML_ErrorString(
            XML_GetErrorCode(p)) << std::endl;
      exit(5);
    }
  }
  // free the parser when we've finished with it
  XML_ParserFree(p);
  std::cout << "Done, \nTotal chars read: "
            << totalCount << std::endl;
  std::cout << "Tags counted: "
            << pec->tagCount << std::endl;
  std::cout << "Attrs counted: "
            << pec->attrCount << std::endl;
  std::cout << "Chars counted: "
            << pec->charCount << std::endl;
  delete pec;
  return 0;
}
```

Listing 2: Basic parsing (file `test02.cpp`)

whenever it encounters various parts of the XML, for instance, start tags, end tags and so on.

To express our interest in these events, we register a set of functions with the parser that it will callback on when a specified event happens. In listing 2 we tell the parser that we are interested in knowing whenever a start tag, end tag or character data is encountered.

We use the functions:

```
XML_SetElementHandler(p, startFn, endFn);
XML_SetCharacterDataHandler(p, characterFn);
```

to register three static functions to handle the callbacks, as usual, passing the pointer to the parser as the first parameter. The other parameters are the function pointers for the start handler, end handler and character data handler.

We make use of a third callback method:

```
XML_SetUserData( p, pec );
```

to tell the parser to pass this pointer to the callback handler functions. This can be a pointer to any data that we want passed to the functions (it's a `void*`), in this case we declare a structure `ElementCounts` to keep track of the tag and character counts that we receive. Note that we are responsible for disposing of the structure when we are finished with it, before freeing the parser.

Now, on to the handlers themselves. The start handler takes the form:

```
void* startFn(void* data, const XML_Char*
name, const XML_Char** attr)
```

The first parameter is the pointer to the user assigned data mentioned above. The second parameter is a pointer to a character array containing the element name and the third pointer is to an array of character pointers to the attributes. `attr[0]` is the first attribute name, `attr[1]` is the value and so on.

The end handler takes the form:

```
void endFn(void *data, const XML_Char *name);
```

with similar meaning to above. If the tag is an empty tag (e.g. `<br/>`) then calls are made to the start and end handlers in order.

The character data handler is a little more complex, it takes the form:

```
void characterFn(void* userData, const
XML_Char* s, int len);
```

Here `s` is a pointer to an array of characters *that are not null terminated.* The number of valid characters is contained in the `len` parameter. Only that number of characters should be copied out and stored for use by the client. And there is no guarantee that this string is the whole string within the element. In fact typically the first call will contain blanks and new lines, the next calls will have the data and the last call contains trailing blanks and new lines. But that cannot be assumed.

```
// Replace the data structure and the three
// event handlers with this:

struct UserData {
  enum { NO_TAG = -1, TAG_USER, TAG_PASS };
  std::vector<std::string> tags;
  UserData() : done(false),currentTag(NO_TAG) {
    tags.push_back("login");
    tags.push_back("password");
  }
  std::string username;
  std::string password;
  bool done;
  int currentTag;
};

static void startFn(void* data, const
        XML_Char* el, const XML_Char** attr) {
  UserData* d = (UserData*)data;
  if(strcmp(el, d->tags[
         UserData::TAG_USER].c_str()) == 0) {
    d->currentTag = UserData::TAG_USER;
  }
  else if(strcmp(el, d->tags[
         UserData::TAG_PASS].c_str()) == 0) {
    d->currentTag = UserData::TAG_PASS;
  }
  else {
    d->currentTag = UserData::NO_TAG;
  }
}

static void endFn(void* data,
                  const XML_Char* el) {
  ((UserData*)data)->currentTag =
                             UserData::NO_TAG;
}
```

```
static void characterFn(void* data,
                const XML_Char* ch, int len) {
  std::string s(ch, ch+len);
  switch(((UserData*)data)->currentTag) {
  case UserData::TAG_USER:
    if(!s.empty())
      ((UserData*)data)->username.append(s);
    break;
  case UserData::TAG_PASS:
    if(!s.empty())
      ((UserData*)data)->password.append(s);
    break;
  default:
    // do nothing
    break;
  }
}

// and in the main body:
// new handler installation
  UserData* pud = new UserData();
  XML_SetUserData(p, pud);
//end new

// ....
ifs.close(); // as before

// new printing code.
std::cout << "User name: <"
          << pud->username.c_str() << ">"
          << std::endl;
std::cout << " Password: <"
          << pud->password.c_str() << ">"
          << std::endl;
// end new

delete pud;
```

Listing 3: Additions to main code

After opening the file for reading, and reading in a chunk at a time, we pass this chunk to the parser in the `XML_Parse` method:

```
XML_Parse(p, buff, len, done)
```

There are four parameters, a pointer to the parser, a buffer to parse, the number of characters in the buffer, and whether this is the final buffer. By passing the number of characters to the function we don't need to ensure that the buffer is null terminated.

The function returns `0` if an error occurred, otherwise `1`.

And that is the basic structure of a program for reading in an XML file and handling the various events that the parser creates. Expat provides a lot of different events, we can provide handlers for all of them, see the header files or the documentation if you're interested. We'll look in more detail at some of them in another article.

I'll extend the program now to make it more practical for our purposes by reading in the user's name and password. The main file remains the same, the changes we will make are in the handlers and in the data structure that we pass around. Listing 3 shows the changes to the handlers and structure (`test03.cpp` in the source package). Replace the three event handlers, define a new data structure and modify the code to print the results.

What we are doing, in short, is keeping track of which tag we are within and, based on that, collecting or ignoring the character data that we are passed.

Some points to note from the code:

1 The `characterFn` function can be called more than once within the same tag, so the characters will be appended to the string until we reach the end of that tag.
2 If we look at the strings as output by `std::cout`, we will see that we also get some of the white space:

```
User name: <
     Jsmith
     >
```

Our program has also appended the new line/carriage return characters to the string. So, in a real world application, we would want to trim the string of extra characters.
3 Using a vector to store the tag names and indexing into it is just one way of keeping track of which tag we are in, there are many others. Using a stack, pushing the current tag onto it, and then popping it off in the end handler is another popular technique. With a large number of tags a `map<string,int>` is a good solution.

Handling attributes

The attributes are passed to the start element handler as an array of `char*`s, the first element of the array being the first attribute name, the next is the value, the next is the second attribute name and so on. The list is ended by a `NULL` entry. In order to keep this article short enough, the online source file (`test04.cpp`) has the details, I'll just give a verbal description here.

A typical means of accessing the attributes is simply to loop through them, like so:

```
for(int i = 0; attr[i]; i += 2) {
   // do something with attr[i] and attr[i+1]
}
```

`attr[i]` and `attr[i+1]` will point to a `XML_Char*` and we will need to make a copy if we want to hang on to them. In our example, we assign them to a string.

Error handling

Errors in an XML file can be broken down into 3 types.

1 System level errors (bad file, disk error and so on).
2 Badly formed XML
3 Non validated XML

System level errors can be taken care of in the normal way, such as checking that the file can be read and so on. Non validated XML errors

will not happen with Expat as it is not a validating parser. That leaves us with badly formed XML errors.

Expat is quite good at returning intelligent parser error strings (in English) or error codes, and there are methods to find the line number, column number and byte offset of the offending byte. (Note that it is a byte offset, not a character offset). An error is indicated when the `XML_Parse()` method returns `0`, in which case the error code and error string methods can be called.

So far we've looked at just a few of the functions in `expat.h`, I'll take a look at some of the other functionality in another article, what we've covered so far is enough to have you parsing XML.

The Xerces Parser

Expat is designed to be small and fast and useable on all platforms, an aim that it achieves but at the cost of a slightly clumsy user interface and only supporting the SAX1 interface. The Xerces parser is at the other extreme, providing SAX1, SAX2 and DOM1 and 2 interfaces, all wrapped in a C++ API. There are language bindings for Java, C++, Perl and MS COM. Like Expat, the library is intended to be cross platform across a wide range of operating systems.

Xerces was a project started by the Apache foundation in 1999 (based on IBM's XML4C) and is still in development. But although still evolving, it is known to be stable and is in use in many applications. Currently (May 2002) the version is at 1.7.0.

Xerces has a different philosophy than that of Expat. Whereas Expat does one thing very well, Xerces aims to provide a full toolkit of XML parsing tools, it supports SAX1 and 2, DOM1 and 2, namespaces and XMLSchema. It is also part of a larger toolkit, hosted at `xml.apache.org`, that includes a wide range of tools for working with XML.

```
// Test of SAX parsing using Xerces C++ parser
#include <util/PlatformUtils.hpp>
#include <sax2/XMLReaderFactory.hpp>
#include <sax2/SAX2XMLReader.hpp>
#include <sax2/DefaultHandler.hpp>

const char* xmlFile =
    "\\localprojects\\c\\sax\\data\\demo.xml";

int main(int argc, char** argv) {

  // initialize the Xerces library
  try {
    XMLPlatformUtils::Initialize();
  }
  catch(const XMLException&) {
    // do something
    return 1;
  }

  SAX2XMLReader* parser =
         XMLReaderFactory::createXMLReader();
  DefaultHandler handler;
  parser->setContentHandler(&handler);
  parser->setErrorHandler(&handler);

  try {
    parser->parse( xmlFile );
  }
  catch(const XMLException&) {
    // do something
  }

  delete parser;

  // And call the termination method
  XMLPlatformUtils::Terminate();

  return 0;
}
```

Listing 4: A barebones program to parse a file with Xerces SAX2 interface

Using Xerces

To understand the differences between Expat and Xerces, we'll do exactly the same in Xerces as we did in Expat. See listing 4 (source `test05.cpp`) for the barebones code to create a parser, read a file and exit (the code does nothing practical). In this example we will make use of the SAX2 interface.

The first difference that jumps out from this code is that the Xerces library needs to be initialized before it can be used, and terminated when it is no longer needed, via calls to the two static methods `XMLPlatformUtils::Initialize()` and `XMLPlatformUtils::Terminate()`. The actual working of the calls will depend on the platform Xerces is built for. (Note, on Xerces V1.5 and earlier there could be one, and only one, call to `Initialize` in an application, otherwise the application would segfault. This has been rectified in the later versions).

A second difference is that we now create our parser using a factory method, `XMLReaderFactory::createXMLReader()`, which returns an instance of the parser (or reader as it is called in SAX2).

Finally we note how the handlers are created. There are three main handlers that the parser makes use of, a document handler for the content of the XML document, an error handler for any errors or warnings in the parse and a DTDHandler. Xerces provides a utility class `DefaultHandler`, that acts as a 'do nothing' class and can be used in place of an actual handler class. By deriving from this we can implement just the functionality that we need.

All in all, a much cleaner interface than that of Expat. To do some useful work in Xerces, the only thing we need to do is provide a document handler class that can handle the events created by the parser, and we do that by inheriting from the `DefaultHandler` class. In deriving from `DefaultHandler` we can choose to override the methods that we need.

`DefaultHandler` inherits from five abstract classes in total (`ContentHandler`, `ErrorHandler`, `EntityResolver`, `DTDHandler` and `LexicalHandler`) but at present we are only interested in dealing with start element, end element and character data events from the `ContentHandler` interface.

Here is the handler class (from `test06.cpp`):

```
class OurHandler : public DefaultHandler {
public:
  OurHandler()
    : charCount(0),tagCount(0),attrCount(0) {}

  void startElement(const XMLCh* const uri,
                const XMLCh* const localname,
                const XMLCh* const qname,
                const Attributes& attrs) {
    ++tagCount;
    attrCount += attrs.getLength();
  }

  void endElement(const XMLCh* const uri,
                const XMLCh* const localname,
                const XMLCh* const qname) {}

  void characters(const XMLCh* const chars,
                  const unsigned int length) {
    charCount += length;
  }

  int getCharCount() {
    return charCount;
  }
  int getTagCount() {
    return tagCount;
  }
  int getAttrCount() {
    return attrCount;
  }

private:
  int charCount;
  int tagCount;
  int attrCount;
};
```

and we use that as the content handler instead of the `DefaultHandler`:

```
OurHandler handler;
parser->setContentHandler(&handler);
parser->setErrorHandler(&handler);
```

Add some code at the end to print out the results and *voila*, the Xerces equivalent to the program we wrote in the Expat section. Because all our handlers are tucked up neatly in a class, there is no need to pass around a separate structure to store the data, it can be part of the class. (See source `test06.cpp`).

You'll notice that we're using the handler as a content handler and as an error handler, this works because the super class, `DefaultHandler`, supplies three do-nothing handlers for the error functionality (`warning(...), error(...), fatalError(...)`), as well as a few other methods. This makes it easier to specialize the class for just the functions that we need. In a full system, you would probably use separate classes for content handling and error handling.

In a similar manner to our example in Expat, we can modify the program to extract some of the data simply by providing a different set of handlers that detect the 'user' and 'login' tags and saves the data. See `test07.cpp` for the details.

The technique is similar to that used in the Expat example, in the start handler, we keep track of which element we are within and in the character handler we collect the strings that we are interested in.

The main difference is in the way the attributes are presented to us: Xerces creates an object of type`Attributes`. `Attributes` is usually implemented as a kind of vector, which contains a list of attribute name/value pairs (`Attributes` itself is an abstract class). These can be retrieved either by index or by name. For example:

```
XMLCh* timestamp = attrs.getValue(0);
```

or:

```
XMLCh* timestamp =
attrs.getValue("timestamp");
```

An `Attributes` implementation will also support a set of other methods, allowing us to find the type of the attributes, the number of attributes and so on.

If you take a look at the code in `test07.cpp`, you'll note that I'm not simply fetching a pointer to a char array. The actual code, in brief, is this:

```
char buff[BUFF_SIZE];
XMLString::transcode(attrs.getValue((int)0),
buff, BUFF_SIZE-1);
```

and this deserves a brief explanation, although it's a complex matter that I'll devote more time to in a future article. You'll remember that I mentioned that Xerces deals with UTF16 encoding internally, and that `XMLCh` is typedef'd to be a `unsigned short` (or a `wchar_t`). However, in our simple examples, we're dealing with plain old character data, so we need to transform it. For this we use the `transcode` method from the `XMLString` utilities, here the result of `getValue()` is transcoded and stored into the buffer. The transcode family of methods are a bit more complex than this quick usage would imply, but more on that at a later date.

Summary

This has been a short tour of the Expat and the Xerces parsers, two of the main SAX type XML parsers available.

The Expat parser has a long and distinguished pedigree, having been created by one of the luminaries of the SGML world, James Clark, and it has been in use in real world applications for many years now. Updates to the code are few and far between, which is a sign that it works well and the bugs have been ironed out of it. Despite a 'C-style' interface, the basic functionality is easy to work with and for simple jobs this is usually the right choice.

Xerces is a parser that is still in development and aims to cover a lot more ground than Expat. It is a fully object oriented design and API and covers SAX1, SAX2, DOM1 and DOM2 APIs. Despite being in development the parser is stable and usable in a production environment, although you may not want to rely on some of the more esoteric functionality without extensive testing.

In this article I've given a brief overview of what is involved in setting up a parser and parsing a simple file. There are lots of online resources that can take you through the next step, of using them in real world applications. In the next article I'll skip the 'intermediate' phase and come back to look at some of the more obscure aspects of parsing XML.

Tim Pushman

Full source code for these programs is available online at (`www.accu.org` | `http://www.gnomedia.com/cw/sax-articles1.0.tgz`)

Features comparison

Interface	Expat	Xerces
SAX1	Yes	Yes
SAX2	No	Yes
DOM1	No	Yes
DOM2	No	Yes
DLL size	135Kb	1,597Kb

References and further reading

[1] Tim Pushman, "A Short History of Character Sets", *C Vu 14.3*
[2] David Nash, " XML Parsing with the Document Object Model", *C Vu 14.5*
[3] Expat Home page: `http://expat.sourceforge.net`
[4] Xerces Home page: `http://xml.apache.org/xerces-c/`
[5] The SAX project: `http://sax.sourceforge.net/`
[6] ExpatPP: `http://www.oofile.com.au/xml/expatpp.html`
[7] C++ Wrapper from Tim Smith:
`http://www.codeproject.com/soap/ExpatImpl.asp`
[8] SAX in C++ from Jez Higgins:
`http://www.jezuk.co.uk/SAX/`
[9] Oxml wrapper (pages in French):
`http://apodeline.free.fr/Oxml/`
[10] LibXML: `http://xmlsoft.org/`

Write for ACCU!

If you would not be forgotten,
As soon as you are dead and rotten,
Either write things worth reading,
Or do things worth the writing.

Benjamin Franklin

What to write?

Here is a small selection of suggested titles. These have been *specifically* asked for by ACCU members. Please look at the list and consider if you can write something on a topic.

- **The preprocessor**
  What does it do? What can I do with it?
- **Working with strings**
  How do they differ in C and C++?
- **Which loop?**
  How do I choose between `for`, `while`, and friends?

Don't let this list constrain what you write! What are you doing right now? What do you know about? Please write something about this for the ACCU journals.

How to submit

You can send submissions by email to `editor@accu.org`. Plain text is perfectly acceptable; there is a Word document template you may wish to use if you want to retain formatting. That's all there is to it – *please write something.*

Pete Goodliffe

Installing and Using MySQL on Windows

John Crickett

This article describes how to install, configure, test and deploy a simple database using MySQL running on a Windows NT (NT4, 2000, and XP) based machine. It is intended to provide the reader with a direct Windows-specific set of instructions on the installation, configuration and testing of a MySQL installation, as the supplied manual is verbose and intermixes different operating systems, making it hard to find what you need right away, and perhaps a little intimidating for those not familiar with Linux or Unix.

MySQL is the world's most popular Open Source Database, designed for speed, power and precision in mission critical, heavy load use. It is maintained by MySQL AB [1]; the company is owned by the MySQL founders. MySQL is available free under the GNU General Public Licence (GPL). Commercial licences are sold to users who prefer not to be restricted by the GPL terms.

Obtaining MySQL

MySQL can be downloaded free (under the GPL), from `www.mysql.com`. This article deals with the Windows version found directly at: `http://www.mysql.com/downloads/mysql-3.23.html` I suggest you download the installation files, and this article will assume you have done so.

Installing MySQL

Once downloaded, unzip the file to a directory, and run `setup.exe`. I suggest you accept the default options for installation, which will install all the required files to `c:\mysql\`. Once the installation program is has finished you'll need to open a console window, and the change to the directory `c:\mysql\bin`, now it's time to verify that you have correctly installed MySQL. Type `mysqld –install`, as shown below to install the server as a service:

```
C:\mysql\bin>mysqld –install
Service successfully installed.
```

Next we need to actually start the service, for this use `net start mysql`:

```
C:\mysql\bin>net start mysql
The MySql service is starting.
The MySql service was started successfully.
```

If we later want to shutdown the service, it is simply `net stop mysql`:

```
C:\mysql\bin>net stop mysql
The MySql service is stopping.
The MySql service was stopped successfully.
```

Now would be a good time to set your computer#s PATH environment variable to include `c:\mysql\bin`, You can do this with the command line:

```
set PATH=%PATH%;c:\mysql\bin
```

You might also want to add this to the end of your `autoexec.bat` (located in `C:\`) or use `Control Panel -> System -> Environment Variables` on Windows 2000/XP, to ensure the path remains set for next time you reboot your PC.

Testing the Installation

We can use the program `mysqlshow` to display the details of the databases in the server (do not forget the restart the service if you have just shut it down):

```
C:\mysql\bin>mysqlshow
+-----------+
| Databases |
+-----------+
| mysql     |
| test      |
+-----------+
```

which lists the databases in this server, or we can get the details of one specific database as so:

```
C:\mysql\bin>mysqlshow mysql
Database: mysql
+--------------+
|    Tables    |
+--------------+
| columns_priv |
| db           |
| func         |
| host         |
| tables_priv  |
| user         |
+--------------+
```

There are number of other useful facilities provided by `mysqlshow`, so use the command line `mysqlshow –help` to see what else is available. Congratulations, you have successfully installed MySQL.

Administering the MySQL Server

To administer the MySQL server we use the program `mysqladmin`, which allows you to perform general administration task on the MySQL server. Such tasks might be:

- Create a database.
- Delete a database.
- Change the admin password.
- Check the status of the server.
- Shutdown the server.

Try the following command:

```
C:\mysql\bin>mysqladmin -?
```

for a full list of available commands; they are all self-explanatory.

Securing the Database

By default MySQL allows all local users to logon to the MySQL server with full privileges, I suggest if you are going to deploy anything more than a toy application that you change this immediately. To do so we use the command line tool `mysql`, which provides a command prompt from which we can send commands and SQL to the server. The process of removing the default access rights is shown below:

```
C:\mysql\bin>mysql mysql
Welcome to the MySQL monitor.  Commands end with ;
or \g.
Your MySQL connection id is 7 to server version:
3.23.49-max-debug

Type 'help;' or '\h' for help. Type '\c' to clear
the buffer.

mysql> DELETE FROM user WHERE Host='localhost' AND
User='';
Query OK, 1 row affected (0.90 sec)

mysql> quit
Bye

C:\mysql\bin>mysqladmin reload
C:\mysql\bin>mysqladmin -u root password secret
```

Here you can see we have used `mysqladmin` to tell the server to reload the user table, thus updating its list of allowable users. This is known as reloading the grants table, as the table "grants" users' privileges.

Of course, I suggest you pick a more secure password than 'secret', for your server. The downside of this is of course the commands we have already learnt now need to be modified to run with your username and password, otherwise you will be seeing results like the following:

```
C:\mysql\bin>mysqlshow mysql
mysqlshow: Access denied for user:
'@localhost' to database 'mysql'
```

So we need to modify the command as so (once again replace `secret` with your password):

```
C:\mysql\bin>mysqlshow -uroot -psecret mysql
Database: mysql
+--------------+
|    Tables    |
+--------------+
| columns_priv |
| db           |
| func         |
| host         |
| tables_priv  |
| user         |
+--------------+
```

Here we are connecting as the user "root" (the `–uroot`), with the password "secret" (`-psecret`), setting our active database to `mysql`.

Creating a Test Database

In this section, we will create a simple example database, and write some simple programs to connect to the MySQL server and query the database. To do this start up `mysql`, as so:

```
C:\>mysql -u root -psecret
Welcome to the MySQL monitor.  Commands end
with ; or \g.
Your MySQL connection id is 9 to server
version: 3.23.49-max-debug

Type 'help;' or '\h' for help. Type '\c' to
clear the buffer.

mysql> create database dev;
Query OK, 1 row affected (0.00 sec)

mysql> use dev;
Database changed
```

We have now created a new database called 'dev', and made it the active database, in other words all the queries we run from now on will be against that database. We can then create our tables, and query the database using SQL, as we would for any other RDBMS. As SQL is beyond the scope of this article, we will just quickly create a few tables and enter some simple data.

I prefer to execute most queries as scripts, especially queries to create tables, indexes and views, populate the static tables or the main tables with test data, and to drop the database. This allows us to create and drop the database as part of an automated build process, making it easy to test the database, with confidence that we can easily restore it to a known, working state. These scripts can now be kept safely in a version control system. We can run a SQL script as follows:

```
C:\dev\articles\mysql\scripts>mysql -uroot
–psecret dev < create_tables.sql
```

where `create_tables.sql` is the SQL script used to create the tables we need (the script is shown in full in the appendix). Note that we can also specify the database to use on the command line.

Connection to the MySQL Database using ODBC

For testing I prefer to use a simple scripting language, as such I use PERL [2] to test that the MySQL ODBC driver is correctly installed and working. The script, `test_odbc.pl` given in the appendix performs the test. Note that you do not have to use ODBC to connect to the database, MySQL comes with its own C++ library, and many open source languages/environments come with support built in.

Installing the MySQL ODBC Driver

I'll assume you've downloaded the ODBC driver from the website, and have unzipped the installer, so now run `setup.exe`. Accept all the defaults, but do not enter a Data Source (just select close on this dialog box). ODBC is now installed, however to use it we will need to configure a DSN, so open up Control Panel, and find the Data Sources icon (Windows 2000/XP users will find in Adminstrative Tools).

Now select Add, then select the MySQL driver, and enter the following values:

Windows DSN name:	`MySQLDev`
MySQL Host:	`localhost`
MySQL Database:	`dev`
User:	`root`
Password:	`secret`

Once again swap `secret` for your password, and then leave the rest as they are, and click OK, and OK.

Running the Test script

Before we can run the test script we need to create some tables in our database, and populate them with some test data, so we will:

```
C:\dev\articles\mysql>mysql -uroot -psecret
dev < create_tables.sql
C:\dev\articles\mysql>mysql -uroot -psecret
dev < populate_tables.sql
```

We are now ready to run `test_odbc.pl`, assuming you've chosen to use ActiveState's PERL distribution we can run it as so:

```
C:\dev\articles\mysql>test_odbc.pl
building_name = ABC House
postcode = SN15 2EX
company_id = 0
county = Wiltshire
building_number =
company_name = ABC Ltd
street = Our Street
town = Chippenham
```

Congratulations: you've installed MySQL and configured the ODBC driver.

John Crickett

References

[1] MySQL AB – `http://www.mysql.com`
[2] Active State Perl – `http://www.activestate.com`

Appendix - Test Scripts

The following scripts are used in the preceding examples. This first script (`create_tables.sql`) is used to create a table called "customers".

```
use dev;
CREATE TABLE customers
(
   company_id INT(10) PRIMARY KEY,
   company_name VARCHAR(200),
   building_name VARCHAR(200),
   building_number INT(10),
   street VARCHAR(200),
   town VARCHAR(100),
   county VARCHAR(50),
   postcode VARCHAR(10)
);
```

The following script (`drop_tables.sql`) can be used to drop our table. This removes the table from the database and will allow us to start over from scratch after we have filled our database with either bad data or test data during development.

```
use dev;
DROP TABLE customers;
```

The next script (`populate_tables.sql`) adds a row to the customers table.

```
use dev;
INSERT INTO customers
(company_id, company_name, building_name,
 street, town, county, postcode)
VALUES
(0, 'ABC Ltd', 'ABC House', 'Our Street',
 'Chippenham', 'Wiltshire', 'SN15 2EX');
```

The final script (`test_odbc.pl`) is a perl script.

```
use Win32::ODBC;
$dsn = "DSN=MySQLdev;UID=root;PWD=secret;";
if (!($db = new Win32::ODBC($dsn))) {
  print "Error connecting to $dsn\n";
  print "Error: " . Win32::ODBC::Error() .
        "\n";
  exit;
}
else {
  $sql_statement = "SELECT * FROM customers";

  if ($db->Sql($sql_statement)) {
    print "SQL failed.\n";
    print "Error: " . $db->Error() . "\n";
  }
  else {
    while($db->FetchRow()) {
      undef %data;
      %data = $db->DataHash();

      while (($column, $value) = each %data) {
        print "$column = $value\n";
      }
    }
  }
}
$db->Close();
```

Effective C++ in an Embedded Environment

Lois Goldthwaite

In embedded programming, every byte of memory and every tick of the clock is a valuable resource to be conserved. Engineers writing code for such environments value frugality over all other virtues. But as embedded systems become ever more complex (think of global telecommunications systems and aircraft flight controls), the abstraction mechanisms and type safety features of C++ offer benefits which can enhance robustness, maintainability, and verifiability of programs.

Scott Meyers, author of *Effective C++*, *More Effective C++*, and *Effective STL*, came to the UK in October to present a series of seminars on 'Effective C++ in an Embedded Environment'. Sponsored by Programming Research, vendor of software tools for quality assurance and code auditing, the talks attracted engineers working on a variety of embedded projects. To illustrate the range, one pair used C++ to program the switching equipment for a large mobile-telephone service provider; another group, just stepping up from C to C++, built equipment to test the handling of automobiles (which is itself controlled by other software).

The first half of the day was devoted to an overview of 'C++ Under the Hood' – a discussion of how C++ language constructs are transformed into object code by compiler and linker. While this is not specifically an issue of embedded programming, Meyers explained, C programmers often shy away from using C++ because of a fear that the language has hidden overheads that add fat and stodginess to their programs. A better understanding of how language features are implemented should help to refute the superstitions, he said.

> 'When I started looking into allegations of "code bloat" I found that everyone knew it existed but no one knew what it was. I want to disabuse you of the idea that C++ cannot be efficiently implemented and therefore is not suitable for embedded programming. With C++, there should not be any additional surprising stack or heap usage compared to equivalent functionality implemented in C.'

No-cost C++ Features

Many features of C++ have a cost only during compilation, if at all. They produce the same object code as if implemented in C. Among these are `struct`s, pointers, free functions – all the mechanisms from the common sub-set of C and C++ – as well as class-static functions and data. The keyword `class` is equivalent to `struct`, and `namespace`s have zero overhead also (well, they may add a few characters to mangled names displayed in a debugger, but have no cost in production code). Calls to nonvirtual member functions and overloaded functions and operators are bound at compile time, so have no runtime overhead.

Meyers classified several additional features as 'no-cost', although sometimes it takes a second look to realise it. These include constructors and destructors, which should contain code for *mandatory* initialisation and finalisation of data structures:

> 'They exist to make sure you can't forget to set up and clean up data. If the code isn't mandatory, don't put it into a constructor or destructor – and if you do, don't blame the language.'

`new` is equivalent to `malloc` plus necessary constructors; `delete` is just `free` plus necessary destructors.

Among the C++ 'no-cost' features Meyers included single inheritance (the C equivalent is a nested `struct`) and, somewhat surprisingly, virtual functions. He analysed at length some typical compiler implementations of inheritance and polymorphism to back up this claim. As a general rule of thumb, any use of virtual functions in a class hierarchy adds a table of function pointers (the *vtbl*) to the code space for each class, and a pointer to this table (the *vptr*) to the memory layout of each object in the hierarchy. Calling a virtual function through a pointer or reference to an object requires indirection through the vptr, then indexing into the vtbl of function pointers, and invoking the appropriate function through its pointer. The pointer indirections consume only a few instructions, but because the desired function pointer is selected at runtime, virtual functions may inhibit other optimisations by the compiler, such as inlining. (However, direct calls to a virtual function, where the type of the object is known at runtime, do allow for optimisation.)

But how do the costs of the virtual-function mechanism stack up against equivalent functionality implemented in C? Pretty well, it turns out. Selecting a function at runtime involves C techniques based on `if`/`then`/`else` or `switch`/`case` blocks, which have performance implications of their own. It may involve adding a type tag field to the `struct`'s data, with a data cost comparable to C++'s vptr. But the C++ compiler handles all the data structures and type-checking automatically, instead of requiring explicit programmer attention. In Meyers's words,

> 'The C code is harder to maintain, harder to get right, and harder to debug. As for performance, I have seen a lot of implementations in C that are larger and slower than C++. I have never seen one that was faster.'

Low-cost C++ Features

The features listed above are no-cost in another sense: if they are not used at all, they are guaranteed to add zero overhead to the program. There are some features, though, which may involve a small cost even if they are not used. Depending on the implementation, support for multiple inheritance, pointers to members, and RTTI (runtime type information) can increase the size of vtbls a bit. And exception handling, even if an exception is never thrown, does add some overhead.

There are two main implementation models for exception handling: one builds data structures on the stack at runtime to make sure that appropriate destructors are invoked if an exception is thrown; the other approach has no runtime impact but uses static data space to manage cleanup if an exception is thrown. If it is certain that no exception is possible in a program, most compilers have an option to disable exception handling support. But dispensing with exception handling is not a 'zero overhead' solution (assuming it is unsuitable just to ignore any error condition that may arise). Other mechanisms, such as returning error codes or setting a global `errno` variable, have their costs as well.

C++'s flexible, customisable `iostream` classes can add a large amount of library code to a compiled program. Unless this flexibility is needed, embedded programmers may prefer to use C-style `stdio`. But I/O is not a major factor in many programs for embedded environments, in any case.

There are some features of C++ that can surprise programmers new to the language, and perhaps this is why superstitions have arisen that C++ generates 'unnecessary' or 'unpredictable' code. Temporary objects are created and destroyed at specific times determined by the source code, although knowing what those times are demands understanding of the underlying C++ semantics and language mechanisms. Passing an object to a function by value, for example, calls its (possibly expensive) copy constructor to create another object, which is destroyed when the function exits. This can be avoided by passing a const reference instead. Inheritance results in implicit calls to base class constructors and destructors, and data members contained within another class are similarly initialised and finalised as part of the lifetime of the containing object. If those functions are defined to be inline, a surprising amount of object code can result from an innocent-looking variable declaration.

Speaking of `inline`, it has both advantages and disadvantages. The popular wisdom is that `inline` functions make code larger but faster; however, this is not necessarily true. For very small functions which are used frequently, eliminating the overhead of setting up stack frames may even cause the overall code size to shrink. But the main benefit of inlining is that it enables optimisers to work their magic on larger stretches of code. On the other hand, if binary compatibility with upgraded versions is important, inlining is not advised.

Templates without Code Bloat

The C++ feature most often mentioned in the same sentence with 'code bloat' is templates. But Meyers maintained that most so-called 'template code bloat' arises from programmers' misunderstanding how templates work, or misusing them. He said flatly,

> 'Templates are not only incredibly useful, but are becoming the primary contribution of C++ to the world. You cannot be an effective C++ programmer in any domain if you don't understand templates.'

The big advantage of templates is that they move overheads from runtime to compile time – traditional object-oriented programming must repeatedly

go through vtbls to recover type information which was known by the compiler but then thrown away.

With most compilers, template functions must be defined in a header, but that does not make them automatically `inline` unless they are inside a class definition. Templates are not really source code; the compiler uses them to generate source code, which it then turns into object code. Code inside a template will indeed be duplicated in every instantiation of the template, but this can be minimised by factoring type-invariant code into a non-template base class or external function. If less capable linkers leave duplicate instantiations in multiple translation units, this can be prevented by explicitly instantiating templates once only.

Somewhat surprisingly, Meyers made no mention of Embedded C++, the subset of Standard C++ proposed by an industry group of (mostly Japanese) companies in the semiconductor business. This subset omits templates, exception handling, multiple and virtual inheritance, namespaces, RTTI, and the `mutable` qualifier, with the rationale that some of those features add unacceptable overhead to programs and the others might confuse embedded programmers.

Low-level C++

Turning to specific issues of embedded programming, Meyers discussed how to tell whether data values in a C++ program can be placed into Read Only Memory (program instructions can always be put into ROM). Variables whose value is known at compile or link time, and whose value does not change, can be put into ROM, although some such objects can be optimised away entirely. Compiler-generated data like virtual function tables and the tables to support exception handling can usually be put into ROM.

Programs for embedded environments often need to read from and write to hardware registers. Typically these I/O registers can be found at fixed locations in a program's address space, and sometimes there are additional status registers whose every bit conveys significant information. Rather than go through low-level bit-twiddling, C++ allows this memory-mapped I/O device to be represented as an object with a meaningful interface:

```
// ControlReg is a four-byte
// (size of int on this processor) register.
// if bit 0 is set, device is ready.
// if bit 2 is set, interrupts are enabled
enum { bit0 = 0x1, bit1 = 0x2, bit2 = 0x4 };
// can go up to bit31 = 0x80000000
class ControlReg {
public:
  bool ready() const {
    return regValue & bit0;
  }
  bool interruptsEnabled() const {
    return regValue & bit2;
  }
  void enableInterrupts() {
    regValue |= bit2;
  }
  void disableInterrupts() {
    regValue &= ~bit2;
  }
private:
  volatile unsigned regValue;
};
```

Qualifying `regValue` with `volatile` indicates that its data may change outside the control of this program and tells the compiler not to optimise away or reorder accesses to the variable. Because the member functions are small and implicitly inline, invoking them should be as efficient as twiddling the bits directly.

Placement `new` is used to position the `ControlReg` object so that `regValue` is reading the correct memory-mapped register. Placement `new` is preferred to a raw cast of an address, because it invokes any constructor that may be defined for `ControlReg` (but any destructor must be explicitly called by the programmer):

```
ControlReg& cr =
   *new(reinterpret_cast<void *>(0xFFFF0000))
   ControlReg;

while (!cr.ready() )
  ;    // wait until the ready bit is on

cr.enableInterrupts();
if( cr.interruptsEnabled() ) ...
```

As an enhancement, making `ControlReg` into a class template would add flexibility – template parameters could be used to specify the correct sized datatype for the underlying storage and the status bit masks, so a single class definition could represent many types of register.

High-level C++

Support for abstraction and encapsulation is the most powerful motivation for moving from C to C++ for complex applications. The ability to create new data types makes it easier to express designs in code. But these abstraction mechanisms need not exact a penalty in size or speed, said Meyers. C++ code can actually be *more efficient* than C. Compiler-generated dispatching of virtual functions is often better than common hand-rolled approximations in C code. The `string` abstraction gives implementations the flexibility to improve performance without changing interfaces: optimisations like reference-counted strings and the 'small string optimisation' outperform `char*`-based strings in some applications. Library design is also evolving in the direction of greater efficiency and flexibility. STL-based techniques like traits classes enable compile-time selection of more efficient algorithms when possible, with fall-back options of not-quite-so-efficient algorithms when necessary.

> 'C++ was designed from day one,' said Meyers, 'to be competitive with C in size and speed. There are well-known implementation techniques that don't waste space and time, and that are at least as efficient as you could implement manually for equal functionality. Compared to C code, C++ never loses and it typically wins.'
>
> ***Lois Goldthwaite***

References

[1] Scott Meyers, Effective C++, Addison-Wesley, ISBN 0201924889

[2] Scott Meyers, More Effective C++, Addison-Wesley, ISBN 020163371X

[3] Scott Meyers, Effective STL, Addison-Wesley, ISBN 0201749629

Linux Server Series Part 2

Choosing Hardware

by Paul Grenyer <pjgrenyer@iee.org>

In this, the second part of my series on setting up a Linux Server, I am going to look at what hardware is required for the server and some possible sources. I'll do this by describing the hardware I've decided to use, where it came from and the problems I had. As the server is intended for home use the emphasis will be on acquiring the necessary parts as cheaply as possible.

The first part of this series dealt with choosing a Linux distribution. I looked at the various different distributions of Linux that people (from `accu-general`) were using, and RedHat and SuSE came out on top as the most popular. I concluded that I would set-up the server using RedHat and then set up a second server with SuSE if there was enough interest.

RedHat Hardware Compatibility List

RedHat maintain a hardware compatibility list [1] on their website where it is possible to search for particular pieces of hardware or generate a list of all hardware compatible with a particular release of RedHat.

When I started looking for hardware for a Linux Server I didn't expect to have any problems. I have installed (RedHat) Linux on a number of different PCs in the past ranging from 200MHz 32MB RAM Pentiums to 1.3GHz 385MB AMD Thunderbirds without any hardware problems (with the exception of incompatible network cards and modems). Now that I've come to put a system together for a particular purpose I had a few more problems, which I finally overcame.

My system consists of the following:

Component	Make / Model
Processor	AMD K6-2 300MHz
Motherboard	GemLight GMB-P56SPC (on-board sound and graphics)
Memory	2 x 64MB SDRAM 133
Hard Disk	20Gb Maxtor (5400rpm UDMA100)
CD Rom Drive	Reveal 4 Speed
Floppy Drive	Sony 3.5 inch Drive
Modem	56k Rockwell External Serial V90
Network Card	Realtek NE2000 PCI
Monitor	IBM Personal System/2 Monochrome Display (approx 10 inch)
Mouse	Generic PS2 3 Button Mouse
Keyboard	Generic PS2
Case	ATX Tower

Choosing Processing Hardware

The processing hardware is the processor, motherboard and memory. The Official RedHat Linux x86 Installation Guide [2] does not state a minimum processor speed or RAM size. I heard on one of the RedHat lists that someone has RedHat 7.2 running on a 486 with 12MB RAM. I tried to install it on a 100Mhz system with 16MB Ram and RedHat 7.2 refused to install due to lack of memory. When I increased the RAM to 32MB it installed without any problems.

I chose an AMD K6-2 300MHz because I happened to already have one. Processors can be picked up second hand from computer fairs very cheaply. The last fair I was at had 200Mhz processors from about £5.

Along with the processor I also had 64MB RAM and I upped it to 128MB by buying a brand new Dimm for about £15. Memory at the moment (May 2002) is still very cheap brand new and probably isn't worth the risk of buying secondhand for the money that would be saved.

The motherboard I picked up second hand at a computer fair for about £20. It came un-boxed with only a drivers CD and no cables. I already had cables, but sets of motherboard cables can be picked up for as little as £10.

Should you need to identify an unknown motherboard, this can be done from the BIOS string which, on AwardBios equipped systems, is normally displayed in the bottom left-hand corner of the screen at start-up under "Press DEL to enter SETUP, ESC to skip memory test." The string from the motherboard I bought is: 09/15/1998-SiS-5598-2AIIG3BC-00

A page on the Phoenix Technologies website [3] explains how to identify the manufacturer from digits 7 and 8 of the penultimate part of the string. In my case the digits 'G3' from ' 2AIIG3BC ' identify my board as being made by GemLight. A search on google should then reveal the manufacturer's website and hopefully you should be able to identify the board and get BIOS updates from there.

In my case [4] it wasn't quite that simple as GemLight have gone out of business and it took me a while to identify the company who had taken over production and support of the board. My point is that if you've bought a cheap motherboard and you need to identify it to upgrade the BIOS, get drivers or manuals etc., it may not be straightforward but it is worth persevering.

Choosing Data Retrieval Hardware

The data retrieval hardware is the CD Rom Drive, Floppy Drive and Hard Disk.

The Official RedHat Linux x86 Installation Guide [2] suggests that any SCSI or IDE (ATAPI) CD Rom drive should be just fine. I've never had a SCSI drive to test. The IDE CD Rom drive I have used for this system was an old one I've had for a number of years and hasn't caused any problems. The more up-to-date drives I've tried have also not given any problems. CD Rom drives can be bought brand new for as little as £23 or used for less from a computer fair.

Any standard 3.5 inch 1.44MB floppy drive should do the job. I picked one up from a computer fair for £3.50. New they cost as little as £8. However, if your motherboard is capable of booting directly from a CD there is no real need for a floppy drive.

The Official RedHat Linux x86 Installation Guide [2] suggests that you need a IDE hard disk, nothing more specific. It also mentions that for a custom installation, such as the type we'll be doing, for a minimal installation you need 350MB and for a full installation you need at least 3500MB. I did a test custom installation with the partitions suggested in the guide for the maximum required space which came to about 7500MB.

Partition	Partition Size
swap	256 MB
/boot	50 MB
/	384 MB
/usr	4000 MB
/var	256 MB
/home	2500 MB
Total:	7446 MB

I originally started with a Western Digital 1.7GB IDE hard disk which I picked up from a computer fair for £15 (although in hindsight it was probably quite expensive for what it was) sharing IDE0 with the CD Rom drive. Unfortunately the combination of motherboard, hard disk and RedHat 7.3 caused an interrupt conflict which prevented RedHat 7.3 from booting prior to installation. If I used a Seagate hard disk or RedHat 6.2 there was no such problem. However the Seagate disk I had was only 100MB and I wanted to use a more up-to-date release of RedHat.

To try and get around the problem I bought a brand new Maxtor 20GB hard disk for about £50. This didn't solve the interrupt problem. I tried upgrading the BIOS, this didn't solve the problem. Eventually I tried putting the CD Rom drive on IDE0 and the Maxtor hard disk on IDE1 and it worked just fine.

Choosing Networking Hardware

The networking hardware is the Modem and Network Interface Card (NIC).

Choosing a modem is very important and you will probably have to pay a bit extra, because the chances are you will have to buy an external serial modem as the majority of internal modems are what are known as WinModems and will not work with Linux. WinModems rely on Microsoft Windows to do some of their processing. This facility is not incorporated in Linux and therefore, generally, you cannot use WinModems. To be absolutely sure you have a compatible modem, buy an external serial modem. Even some USB modems rely on Windows to do some of the processing. I bought a brand new external serial modem for about £25.

The choice of Network Interface Card is also very important. I have tried

several different NICs with Linux, including SMC Ultra and D-Link cards, without any success. The only cards I've found to work are NE2000 compatible cards. I'm sure these are available cheaply at computer fairs, but as an adequate 10MBs combo card brand new is only £8 I bought two (one for my client PC as well). Combo cards come with both a BNC and RJ45 connector. I'm using the BNC connector as I have plenty of ready-made cable and a hub (approx £80 new) is not required for more than two PCs.

Choosing Human Machine Interface Hardware

The man machine interface hardware is the mouse, keyboard and monitor. I'll also include the case here as there is no point in putting it in a category of its own.

As this is a server, after installation and configuration of a SSH (Secure Shell, an alternative to Telnet with encryption) server, a mouse, keyboard and monitor should no longer be needed as the server can be completely controlled via an SSH client.

The mouse will only be used during the actual installation as a server does not require X Windows (more formally, the X Window System) or any of the Window managers, such as KDE, which sit on top of it. If you would prefer not to use a mouse at all there is a 'text' installation method. RedHat 7.2 should support any serial or PS2 mouse. I suspect it may also support USB mice during installation. I'm using a standard 3-button PS2 mouse. Brand new mice start from around £5.

RedHat 7.3 will also support any AT or ATX keyboard. Again I suspect it probably supports USB keyboards during installation. I'm using a standard generic PS2 keyboard. Brand new keyboards start from around £8.

You should be able to use any VGA or SVGA monitor with RedHat 7.3. I am using an old monochrome monitor as I happened to have it spare. There is probably no point in buying a monitor for the server as, mentioned above, it is unnecessary to use one after installation. If you do decided to buy one, basic monitors at computer fairs start at about £20, reasonable monitors start at about £40 and brand new 15-inch monitors start at about £80.

A case obviously doesn't need to be anything specific for RedHat 7.3. Make sure you have enough room for all the hardware you need to put in it and that you have an AT case for an AT motherboard and an ATX case for an ATX motherboard. Also make sure that if you have a Pentium III or better or an AMD Athlon or better processor and board, that the case has a 300W power supply. You are unlikely to find one at a computer fair second hand. New ones start from about £20.

In this second part of my Linux Server series I have described the components needed for a Linux Server and discussed cost and possible sources. I have assumed that the reader has enough knowledge to build the PC and construct the cables. In the next part of this series I will be looking at installing RedHat 7.3 and setting up SSH for remote access from a client PC on the network.

Paul Grenyer

Price Summary

Component	Computer Fair Price	New Price
Processor	200MHz from £5	AMD Duron from £35
Motherboard	Socket 7 300Mhx from £20	Socket A from £80
Memory	Buy New (£15)	64MB from £15
Hard Disk	1.7GB £15	20GB £50
CD Rom Drive	Buy New (£23)	£23.00
Floppy Drive	£3.50	£8.00
Modem	Buy New (£25)	£25.00
Network Card	Buy New (£8)	£8.00
Monitor	From £20	£80.00
Mouse	Buy New (£5)	£5.00
Keyboard	Buy New (£8)	£8.00
Case	Buy New (£20)	£20.00
Total:	£167.50	£357.00

References

[1] RedHat Hardware Compatibility List:
`http://hardware.redhat.com/hcl/`

[2] Official RedHat Linux x86 Installation Guide:
`http://www.redhat.com/docs/manuals/linux/`

[4] Phoenix Technologies Motherboard manufacturer's page:
`http://www.phoenix.com/pcuser/phoenixbios/motherboard.html`

[5] Details of the manufacturer and the upgrading of my GemLight motherboard can be found here:
`http://www.paulgrenyer.co.uk/articles`

Thank You. Thanks to Tim Pushman for his continued support throughout this project.

Self-Documenting Code

by Hubert Matthews <hubert@oxyware.com>

Given the recent heated interest in template metaprogramming to use the compiler as a compile-time interpreter, I decided to see if I could get the compiler not only to compile my program but also to write the documentation for it too. To this end, I offer the following 4-line self-documenting C++ program for use with the command-line version of Microsoft Visual C++ version 5:

```
garp.cpp
garp.cpp(1) : error C2501: 'garp' : missing
      decl-specifiers
garp.cpp(1) : error C2239: unexpected token
      '.' following declaration of 'garp'
garp.cpp(1) : error C2059: syntax error : '.'
```

Users of gcc 2.96 will no doubt be pleased to know that the corresponding program is a one-liner:

```
foo.cpp:1:35: missing terminating ' character
```

Naturally, wishing to develop the program using Extreme Programming, I tried to write the tests first. This proved to be difficult because of the cyclic dependencies inherent in the domain, leading me to a truly iterative development approach involving machine-generated code and test results.

Others might like to find more "efficient" and compact versions of this program. Newton-Raphson iteration would be difficult, given the need to differentiate the transfer function of the compiler (= d/dt ISO/IEC 14882) and account for the attendent QoI noise, and we would also need to decide on the level of tolerance (both machine and human) required for termination. Perhaps the fractal nature of the problem will lead to brightly coloured Mandelbrot-style pictures showing the variation of the stability of the program over a range of possible input modifications. Bistable programs with built-in persistence are also possible, even if the data density and bandwidth are rather low. It is unclear how this might interact with version control systems. Embracing change and "empowering the compiler" by allowing it to influence its working conditions are also all very topical human resource issues. My impression is that the inclusion of chaos theory into most software processes would, however, go unnoticed.

Given the compile-time nature of the program, it can offer the strong exception safety guarantee, assuming that the compiler doesn't leak resources and remains in a usable and uninstallable state if it (quite justifiably) throws during compilation. Thread safety is another issue as there appears to be no way short of file-level locking to avoid concurrent modifications of the input program from interacting. Users should therefore avoid the use of parallelised versions of make with this program. Use of the technique for regression testing the compiler is also a possibility.

Related challenges are the "who can get the most error messages from the smallest program" puzzle, and its corollary: "who can get the fewest error messages from the largest program". This last one is seemingly particularly onerous for some programmers.

Given the self-documenting nature of the code, I believe comments are unnecessary. Adding them is left as an exercise for the reader, as is printing out a "this page is intentionally blank" header.

Disclaimer: The above programs have not been checked for viruses or any other forms of self-replicating code. The author denies any responsibility for the content of the code, including any claim to the intellectual property allegedly contained therein.

PDF Problems - Can We Learn From Them?

Silas Brown <ssb22@cam.ac.uk>

I have recently been struggling to print some PDF files. Since PDF is supposed to be a highly portable document format that many readers will be familiar with, I thought I'd describe my cases to see if we can learn anything from the way the system failed.

Adobe's Portable Document Format (PDF) is basically a compressed version of PostScript, which is essentially a programming language that outputs pages of graphics. Adobe's Acrobat Reader decompresses and interprets the PostScript and displays it on the screen or printer. It doesn't always do a very good job of on-screen display unless you've been careful about what sort of fonts you use, but I'll ignore that for now and concentrate on the printing problems.

My first problem began when a Chinese friend wanted to renew her visa, and needed to print a form which was stored in PDF format on the Chinese Embassy website. She claimed that the PDF file was corrupted, but that the Embassy were very unhelpful when she contacted them about it. So she asked me to repair the corrupted file.

Actually, the PDF file was not corrupted, but it required Acrobat Reader version 5 and the Chinese language pack. The university computers only had version 4, and establishments of this size tend to take a while to upgrade their systems; anyway, Acrobat 5 was at that time unstable (especially the Unix version) and I ended up resorting to screen shots in order to get the thing printed.

The first problem here is that the user was given no clear message to the effect that a later version of Acrobat was required. The PDF format does carry a version number, so why didn't Acrobat 4 just tell you when you need a later version, rather than print up some obscure error message that leads people to think that the file was corrupted, or they did something wrong, or something? I think it's because Adobe forgot to change the number stored in the PDF file; they just added more features. That shouldn't have happened.

Secondly, the software that was used to produce the PDF file should have made it clear that the file it was producing needed version 5 and the Chinese language pack; then at least the Embassy would be aware of this. Also it should have had an option to produce a backward-compatible file; it is true that this would be larger because all the Chinese characters would have to be embedded as graphics, but I'd have thought it would be a good idea in some cases, especially if you're targetting people in British universities (which don't always have the latest versions of everything and which don't always have Chinese support). I always manage to make Chinese PDFs that do not require any special add-ons, so I know it can be done.

My second case involves the Times New Roman font. A friend in Germany made a PDF file that mixed Times New Roman with some other font that contained unusual accents; he used Times New Roman for all the non-accented characters, and the other font for the characters with unusual accents. The resulting typesetting was a little unusual but it was good enough.

The problem came when this file was printed. Printing with Acrobat 4 led to rather a lot of characters being printed on top of each other, even though it looked fine on the screen. What had happened was that Acrobat substituted some PostScript font for Times New Roman, and the PostScript font that it substituted didn't have quite the same metrics, i.e. some of the characters were slightly wider or narrower than Acrobat 'thought' they were. If the document were entirely in Times New Roman then this might not have been noticible, but because this font was tightly mixed with another, embedded one (a specialist font that could not be replaced with a standard PostScript one), there were collisions (characters printed on top of each other).

The only way around this problem using Acrobat 4 was to print the file as graphics, but for some reason this resulted in much reduced quality. Acrobat 5 did something better, but again the Unix version was very unstable, and I had to do all kinds of tricks to get it to print without crashing (such as putting the file through pdflatex, adding spare blank pages at the front, and running Acrobat on different sorts of X server); I tried things almost at random until it worked.

Of course, the document's author seemed completely unaware of the fact that he had produced a PDF that required version 5 and that didn't print properly on earlier versions, and was therefore less portable because Acrobat 5 was not stable (at least at the time of writing) on so many platforms as Acrobat 4 is. The principle of warning the user about version dependencies applies here too. And I'm not convinced that this model of font substitution is a good idea; why couldn't the software that made the PDF have used fonts that didn't have these complications? It's probably some horrible artifact of Windows (this kind of thing doesn't tend to happen when you produce documents in the Unix world).

Am I being unfair to Acrobat by picking out obscure special cases? I don't think Chinese is that obscure (from a global viewpoint), and it is this kind of thing that causes the need for a graphics-oriented distribution format in the first place. After all, if you are sending text in a character set that you know the target system supports, then you might as well send it in a malleable, accessible format like a text file, unless you want to produce camera-ready copy for a publisher whose typesetting you don't trust. A major reason for using a graphical format is that the target system does not support the character sets you want to use, so I'd have thought that any graphical distribution format should have this as a test case.

One final thing: Adobe have a website that allows PDFs to be converted to HTML online, for the benefit of users with reading difficulties who need to get the text into an alternative format. This is nice, even if it doesn't always work (in particular it doesn't work with ligatures), but sometimes it complains that it cannot convert a document because the document's security settings prohibit conversion. I wonder if the authors of such documents realised that they were choosing security settings that had the side-effect of preventing blind people reading their documents? They should probably have at least been informed of this in the security dialog box. Security can be a touchy subject; if I contact an author and ask them to change the security settings they sometimes think I'm trying to pull off a computer crime or something (if they have the time to deal with my request at all). Such is life in an ill-understood minority...

Uninitialised Variables in C: What to Expect

Victoria Catterson <vic@cowlet.org>

Different things can be expected of uninitialised variables, depending on how they were declared. Broadly speaking, there are three different sorts of variables: static, and automatic variables, and dynamically allocated memory. Each type will be uninitialised in a reliable, if not always useful, way.

Local and global variables can be of the static storage class. Local static variables retain their value between function calls (even though they have only block scope), and are declared with the `static` keyword. Global variables are always static. If a global variable is declared with the `static` keyword it has file scope, and is accessible only to functions within the same file. If a global variable is declared without the `static` keyword it has program scope, and is accessible to functions throughout the whole program. Even though the `static` keyword is not used, program scope variables are still of the static storage class. All static variables are initialised to zero [1]. They are also guaranteed to be the correct type of zero, such as `0`, `NULL`, `0.0`, or `{0}`.

Uninitialised automatic variables are not as simple. Local variables declared without the `static` keyword default to the automatic storage class. Because they are allocated stack space when defined, they contain whatever was previously on the stack at that location. This is unlikely to be anything useful, so uninitialised automatic variables can be expected to contain garbage [2].

The contents of memory which is dynamically allocated are also unknown [3]. One exception to this is when the memory is allocated with `calloc`, which sets all the allocated bytes to `0`. However, care should be taken with this, as `0` is not necessarily the same as `NULL` or `0.0`. For this reason, it is safest to use `calloc` only for integers or strings.

While it is useful to know what to expect from an uninitialised variable, it is unadvisable to use variables without initialising them first. Certainly, automatic and dynamic variables cannot be relied on to contain particular values, and so must be initialised before anything useful can be done with them. However, knowledge of what uninitialised variables look like can be helpful when debugging. If a variable contains a really odd, unexpected value, check it has been initialised correctly!

References

Kernighan and Ritchie, *The C Programming Language*, 2nd Edition,
[1] Section 4.9, Initialization, pg 85
[2] Section 1.10, External Variables and Scope, pg 31
[3] Section 7.8.5, Storage Management, pg 167

Python Section

Python And C Vu

Paul Brian <paul1brian@yahoo.com>

Welcome back to the Python section of C Vu. In this edition we have an article by Richard Taylor that I feel clearly demonstrates three of Python's most powerful features - how well it is supported by third party add-ons such as its smooth support for almost any database, its ability to expose its internals and change them on the fly, and touching upon list comprehensions, a nod towards the functional programming that is buried inside the language.

Like many of the best loved parts of working life, these language features are rarely encountered and rarely needed, but when they are needed, they are so useful, so right, so easy, that they provoke a pleasing feeling of (self-)satisfaction. And perhaps a sigh of relief for having avoided doing things the hard way.

For me, these useful little language tools remind me of why I got into computing in the first place. Something of the basic drive to at least understand what shapes our world, just to feel that the inexplicable is just a bit more explicable, a bit more under our control, another tool stowed away neatly for later use. Perhaps an echo of how it felt to discover rubbing two sticks can produce an ember - unexpected, useful and satisfying.

But enough waffling: please read on, try out the code and dip into Python. Even if it does not drag you back to why you started all this, I hope it inspires you to add another notch to your bow.

Look forward to seeing you in the next edition.

Paul Brian
Sort of Python Editor

Using Python's Dynamic Features to Encapsulate Relational Database Queries

Richard Taylor

Introduction

Connecting to a relational database is one of the most common things that programmers expect any programming language to support. In this article I hope to show how some of Python's unusual dynamic features can be used to provide a flexible interface to a relational database.

Python is well equipped with libraries which give access to many RDBMSs, including Oracle, DB/2, Informix, PostgreSQL and MySQL as well as interfaces to JDBC[1] and ODBC[2]. To ease the task of the developer when moving an application between different databases, a group of Python users and database module developers have collaborated on the specification of a standard database access API known as the DB-API, currently at version 2.0. Using the DB-API it is relatively simple to port a Python program from one RDBMS to another, assuming that any database features that the application uses are available on the target database, of course.

The DB-API v2.0 has been around since 1999 and an introduction to it can be found on the DB-API Special Interest Group at the Python web site [3]. Rather than repeat a detailed guide to the DB-API, which can also be found on the web site, this article will concentrate on the way in which a simple object wrapper can be layered on top of the DB-API. This wrapper provides an abstraction from the simple SQL procedural interface provided by the DB-API. My own team uses this approach in our database code because it provides some isolation from changes to the database schema and places all of the SQL syntax in the one place. It also makes the main application code more "Pythonic" by wrapping the database records with objects.

This approach demonstrates how Python's dynamic interpretation features can be used to make an abstract interface that is highly flexible.

Using the DB-API

Listing 1 shows a simple example of using a DB-API database adapter, in this case it is the PostgreSQL adapter `psycopg`[4]. `psycopg` is tuned for highly multi-threaded applications and has proved very reliable in the applications in which we have used it. The example code here should be portable to any of the DB-API compliant modules, with the possible exception of the connection string which is likely to be specific to each database adapter.

This example creates a single table, inserts a couple of rows and then pulls them back out. The DB API adaptor returns the result set as a list of tuples. The final line is the output from running the script. You can see from this example that the Python print statement understands how to turn Python data structures into human readable form. This is very useful when debugging an application, particularly because Python's dynamic type system means that you can be unsure of the type of the variable that is being printed.

```
import psycopg

cnx = psycopg.connect("user=postgres dbname=db_test")
cr = cnx.cursor()
cr.execute('create table telephone ("name" text, "tel" text)')
cr.execute("insert into telephone values ('Richard', '12345678')")
cr.execute("insert into telephone values ('Steve', '01002030')")
cr.execute("select name, tel from telephone")
cnx.commit()
result = cr.fetchall()

print result

[('Richard', '12345678'), ('Steve', '01002030')]
```

Listing 1 - Simple DB API example

Wrapping the records

The low-level API used in Listing 1 provides all the functions required to store and retrieve data in a database. However, littering an application with literal SQL strings and duplicate code to unpack the records from the return list can be tedious. In large applications it can also lead to poorly maintainable code that is highly susceptible to changes in the database schema.

The `Record` class in Listing 2 provides a object that can represent one of the tuples in the return list from the DB-API "fetchall" function. The class constructor uses the description provided by the DB-API to dynamically construct an object that appears to have an attribute for each of the fields in the database record. This is achieved through the use of the special `__getattr__` method. Python classes have a small number of special methods which, if overloaded can change the way an object behaves. `__getattr__` is called by the interpreter when an attempt is made to access an undefined attribute of an object. This is possible because Python objects hold references to their attributes and methods in a special attribute called

```
class Record:
  """A simple class used to wrap the database records
  returned from a DB-API compliant database module"""

  def __init__(self, description, values):
    """The record constructor.

    The description parameter should be the contents
    of the cursor.description attribute after the call
    to cursor.execute(). The values parameter should
    be a list of field values in the same order as they
    are returned from the cursor.fetchXXX() call."""

    # The cursor.description attribute is a list of
    # tuples, where the first element of each tuple is
    # the field name. To make the name lookups a
    # little easier a list of field names is extracted
    # from the description.

    field_names = [ d[0] for d in description ]

    # A dictionary of (field name, field value) pairs is
    # constructed, keyed on the field name. This is then
    # used by the __getattr__ and __setattr__ methods
    # to perform the field lookups. This must be
    # explicitly added to the __dict__ dictionary in
    # order to shield it from the __setattr__ method
    # defined below.

    self._values = {}
    for field_num in range (0,len(field_names)) :
      self._values[field_names[field_num]] = \
           values[field_num]

  def __getattr__(self, attr):
    """This overrides the attribute access method to
    look up attributes in database field names if they
    are not found in the objects dictionary first."""

    if self._values.has_key(attr):
      return self._values[attr]

    raise AttributeError

  def __setattr__(self, attr, value):
    """This overrides the attributes assignment method
    to ensure that any assignments to database fields
    are applied to the _values dictionary."""

    if self.__dict__.has_key('_values') and \
      self.__dict__['_values'].has_key(attr):
      self.__dict__['_values'][attr] = value
    else:
      self.__dict__[attr] = value
```

Listing 2 - A simple Record class

`__dict__`. The interpreter interrogates this dictionary when an attribute or method is accessed and if no match is found the `__getattr__` method is called. The `__dict__` dictionary can be manipulated directly at runtime to change the behavior of the object. Careful examination of the code in Listing 2 will reveal how the `__dict__` attribute and the `__getattr__` method can be combined to dynamically give an object attributes that correspond to the columns returned by a database query. Listing 3 shows how this new `Record` class can be used to encapsulate the records returned from the database.

The `Record` class in Listing 2 also defines a `__setattr__` method. This is another of Python's special methods: it enables assignment to attributes to be as dynamic as attribute access. However there is an important subtle difference between `__getattr__` and `__setattr__`: whereas `__getattr__` is called only when no matching attribute can be found, `__setattr__` is always called first and only if `__setattr__` raises an exception will the interpreter search

```
cr.execute("select name, tel from telephone")
cnx.commit()
records = []
for row in cr.fetchall():
  records.append(Record(cr.description, row))

for record_object in records:
  print "Name = %s Tel = %s" % (record_object.name, \
         record_object.tel)
```

Listing 3 - Using the Record class

```
class View:
  """A class to manage access to a database view."""

  def __init__(self, tablename, record_class):
    """The view constructor.

    The tablename parameter should be a string holding
    the name of a table that has already been created
    in the database. record_class should be a class
    object that provides the same constructor interface
    as the Record class below. """

    self.tablename = tablename
    self.record_class = record_class

  def fetchall(self, cr):
    """Accessor method. Returns all of the records
    in a table.

    The return value is a list of record_class
    objects populated with the table rows returned
    from a select * query."""

    cr.execute("select * from %s" % self.tablename)

    return [ self.record_class(self,cr.description, \
      row_values) \ for row_values in cr.fetchall() ]
```

Listing 4 - A simple View class

the rest of an object's attributes. The alert reader will spot that this could easily lead to an infinite recursion if the body of the `__setattr__` method accesses an attribute of "self". To alleviate this problem access to the special `__dict__` attribute does not cause a call to the `__setattr__` method. This one special case enables `__setattr__` to be used effectively, if with some care.

Having provided the ability to assign new values to the database fields of the `Record` class the obvious next step is to be able to write the updated values back to the database. Listing 4 introduces a `View` class to manage the information needed to construct update queries and provide select query methods for a table.

The `View` class is fairly straightforward but there are a couple of points worth noting. First, the `record_class` parameter to the `__init__` method is a class object not a class instance. In Python class definitions are first class objects themselves and can be passed as parameters just as their instances can. This enables a type of generic programming. In the `View` class the `record_class` object is used to instantiate the record objects in the `fetchall` function. Second, the construct used in the return statement of the `fetchall` function, denoted by the square brackets, is known as a list comprehension. List comprehensions are a convenient way of constructing anonymous lists and can be used as an alternative syntax to the functional programming functions `map` and `filter`[5]. A full explanation of list comprehensions can be found at [6].

Listing 5 shows a new version of the `Record` class that makes use of the `View` class to implement an update method. The `__getattr__` and `__setattr__` methods have been omitted because they are identical to those in Listing 2. The `update` method writes the record fields back to to the database. It is worth noting the use of `copy.deepcopy` in the `update` method. As you would expect, simply assigning a dictionary to

```
class Record:
  """A simple class used to wrap the database records
  returned from a DB-API compliant database module"""

  def __init__(self, view, description, values):
    """The record constructor.

    The description parameter should be the contents of
    the cursor.description attribute after the call to
    cursor.execute(). The values parameter should be a
    list of field values in the same order as they are
    returned from the cursor.fetchXXX() call."""

    # Remember the view object reference.
    self.view = view

    # The description attribute is a list of tuples,
    # where the first element of each tuple is the field
    # name. To make the name lookups a little easier a
    # list of field names is extracted from the
    # description.
    self.field_names = [ d[0] for d in description ]

    # A dictionary of field name, field value pairs is
    # constructed, keyed on the field name. This is then
    # used by the __getattr__ and __setattr__ methods
    # to perform the field lookups. This must be
    # explicitly added to the __dict__ dictionary in
    # order to shield it from the __setattr__ method
    # defined below. A second copy is created to be used
    # in the where clauses of update queries.

    self._values = {}
    self._original_values = {}

    for field_num in range (0,len(self.field_names)) :
      self._values[self.field_names[field_num]] \
        = values[field_num]
      self._original_values[self.field_names[field_num]] \
        = values[field_num]

  def update(self, cr):
    """Write back changes made to a record to the
    database.

    The cr parameter should be an open cursor object and
    it is the responsibility of the caller to ensure
    that commit is called on the cursor if autocommit
    mode is not used."""

    s = "update %s set " % (self.view.tablename,)

    new_field_values = []
    for field in self.field_names:
      new_field_values.append("%s = '%s'" \
        % (field, self._values[field]))

    s = s + string.join(new_field_values,' , ') + " \
      where "

    old_field_values = []
    for field in self.field_names:
      old_field_values.append("%s = '%s'" \
      % (field,self._original_values[field]))

    s = s + string.join(old_field_values,' and ')

    cr.execute(s)

    self._original_values = \
      copy.deepcopy(self._values)
```

Listing 5 - Record class with with update method.

```
# Create a new cursor
# (assumes the lead in code from Listing 1).
cr = cnx.cursor()

view = View("telephone", Record) # Create a view
               # object for the telephone table.

records = view.fetchall(cr) # Fetch a list of all
               # records in the telephone table.

records[0].name = "new name" # alter a field in the
                             # first record
records[0].update(cr) # write altered record back to db.
```

Listing 6 - Putting it all together.

a new name only creates a new reference to the dictionary. `copy.deepcopy` creates a new dictionary with a copy of each element of the original.

Listing 6 demonstrates how the `View` and `Record` classes can be put together to access and manipulate the database records of a table without ever explicitly stating what the fields of the record are. Additional fields can be added to the tables without requiring any alteration to the classes and no table specific SQL sytax is required in the main application code.

Taking it further

By sub-classing `View` and `Record` it is very simple to add more complex capabilities. For instance calculated fields could be added to records or complex query methods added to views. By encapsulating business logic in subclasses of `View` and `Record` it is possible to ensure that such logic is as independent of the details of the database interface and schema as possible. One technique that I have used in my applications is to insert name mapping into the `__getattr__` and `__setattr__` methods when a database field name changes. This can enable such changes to happen without altering any of the business logic that would otherwise have to be kept in step.

The implementations of `View` and `Record` presented here are obviously missing many useful features such as dealing with field types other than strings, views that are made from joins across multiple tables, inserting new records into tables etc. There is also no attempt to catch any errors. However, hopefully I have shown how Python's highly dynamic nature might be used to add all of these features in a very flexible manner.

Final thoughts

This article has demonstrated how features such as dynamic attribute lookup and class definitions as first class objects can be used to build flexible abstractions in Python. I have used such techniques extensively in the applications on which I have worked. However, I have also learned through bitter experience that such techniques can cause faults that, because Python has little compile time checking, only become apparent at run time and can prove very hard to find. These problems can be alleviated with judicious use of pre- and post-conditions on methods along with careful use of exception handlers to recover from runtime errors.

Reading back through this article it also occurs to me that I may give the impression that Python is full of special case rules and littered with built-in method names. This is not the case. There are in practice only a small number of special cases to learn and it is extremely rare to be tripped up by a built-in method that you have overloaded by accident. Most of the time Python can be used in blissful ignorance of the machinery that enables the dynamic features that I have used in this article.

Richard Taylor

References

[1] JDBC: `http://www.ziclix.com/zxjdbc/`

[2] ODBC Module: `http://www.python.org/windows/win32/odbc.html`

[3] Python Database API Specification v2.0: `http://www.python.org/peps/pep-0249.html`

[4] Psycopg Home Page: `http://initd.org/software/psycopg`

[5] Functional programming in Python: `http://www-106.ibm.com/developerworks/library/l-prog.html`

[6] What's New in Python 2.0: `http://www.amk.ca/python/2.0/`

Reviews

Bookcase

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

Francis Glassborow writes:

The number of books available for review is about the same now as it was two months ago. In other words the inflow from publishers about balances the outflow to reviewers.

Several publishers now send me everything that they publish in the general programming and computing area (one even sends me the medical books they publish because getting the distribution department to distinguish has proved impossible.

How long a book stays on my review shelves waiting for a volunteer depends on how relevant I think the subject matter may be. An unsolicited book on Visual Basic is likely to be handed on to Oxfam much faster than a book on a programming methodology.

I have quite a lot of books on various aspects of games programming that are looking for reviewers. Perhaps we lack the number of enthusiastic teenagers that ACCU used to have and so there are fewer people interested in this area. Perhaps that is something that members might think about. The young (or old, when it comes to that) enthusiast has always been welcome in the ranks of ACCU. Perhaps some of you might think about the young generation and consider giving them a membership of ACCU as a Christmas present. I know of several current active members whose first membership was a gift from an uncle, aunt or older sibling.

I would also like to encourage reviewers to consider writing longer reviews for our website together with condensed reviews (i.e. about the current size) for printed publication.

If every member found time to do a review every couple of years I would have more space on my office floor. You would also find that you became more critical of the books that are published.

Well have a good Christmas and enjoy your reading. Do not forget to write in if you think one of our reviews is unfair (either to the author by being unjustly critical or to the reader by having been not critical enough). I note that one of the books that the review below does not recommend got 5 stars in no less than three Amazon reviews (a fourth person questions whether the reviews are genuine, which in view of our reviewer's opinion seems to be a point that should be considered).

Francis

<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

C++

Object-Oriented Programming in C++ by Nicolai M. Josuttis (0-470-84399-3), Wiley, 610pp @ £29-95 (1.67)
reviewed by Francis Glassborow

When this book hit my desk a sinking feeling went through me because I know the author well and have a very high regard for his writing and technical knowledge. The reason for my reaction is that, as quite a few of you know, I have finally got down to writing the book for novices that I have long promised. I feared that Nico might have stolen the very readership at which I am aiming. I tell you this because I think you are entitled to know that I might not be entirely free from bias.

What worried me was the following paragraph from the Preface:

> The result is a book for all beginners who want to learn and understand how to program in C++ as well as those programmers who want to get the overall picture and take advantage of the standardized C++ language and its standard library.

I had not read very far into this book before I relaxed and realised that I had nothing to worry about. This quotation under the heading 'Prerequisites' in chapter 1 makes it clear (at least in my view) that the above paragraph is an excessive claim:

> The principal prerequisite for understanding this book is a degree of familiarity with the concepts of higher-level programming languages. It is assumed that the reader already knows terms such as *loop*, *procedure* or *function*, *statement*, *variable*, etc.

I do not think any reader can be familiar with such terms and still be a beginner at programming. They might well be newcomers to Object Orientation and/or to C++, but anyone with such familiarity is surely not a newcomer to programming.

In fact this is a well-written, technically correct but very traditional book on OO with C++. Indeed, I would question the OO claim, as it is very much a book about C++ programming. Another claim that I think highly debatable is found under *How Should You Read this Book*:

> As this book introduces all language features and their applications gradually, beginners should simply read the book from cover to cover.

Note that, for example, the author covers templates in 44 pages. That may well be adequate for an experienced programmer who already has some grasp of generic programming, but in my opinion such coverage has little place in a book aimed at genuine newcomers. Note that the author claims to give complete coverage of C++ (which, of course, he does not) in a book whose page count is 70% of Bjarne Stroustrup's *The C++ Programming Language'* 3ed. And the pages are less densely packed with text (which makes the book less daunting) so I guess that the book is actually about half the size.

The book originated in German in 1994, and went to a second and substantially revised edition before being translated to English. At the time of the original German first edition I guess that it was a much better presentation of C++ than anything then available in English (the main contenders would have been the second editions of Stroustrup's *The C++ Programming Language* and the second edition of Stan Lippman's *C++ Primer*. However putting a translation phase into this book means that it now lags (in my opinion) behind such works as Koenig and Moo's *Accelerated C++*.

There is another weakness in using this book as a study text; it has no exercises. Novices need exercises to help them consolidate on their learning.

Perhaps you now understand why I felt it essential to declare my interest before launching into this review.

This book is a technically solid, excellently written introduction to C++ for those with more than a little programming expertise in one or more other languages. It would make a good text to accompany a course on C++ but I do not think that it is adequate for the solo student trying to learn to program in C++. Despite the author's claim it is fast paced and covers an immense amount. I think he probably does not recognise just how much ground is covered and just how much demand it places on the reader's prior experience.

On thinking about it, I have reached the conclusion that the ideal reader will be one who has acquired a smattering of knowledge of C++ from other sources (such as bad courses, poor books etc.) and knows that s/he needs to start afresh to correct and consolidate their knowledge and understanding. That, of course, means that it has a vast potential readership. Regretfully many of them will not recognise that this book is exactly what they need to remedy the damage that has been done to them by earlier misguided efforts to learn C++. I think I might well suggest that those struggling with university courses purporting to teach them C++ might be particular beneficiaries of this book. There the lack of exercises and relatively high pace would be advantageous. For the rest, first time students of C++ with a programming background I would still recommend *Accelerated C++*. Those who know nothing about programming will have to be patient.

C# & Python

Practical Python by Magnus Lie Hetland (1-59059-006-6), Apress, 618pp @ £32-50 (1.54)
reviewed by Francis Glassborow

As many of you will have noticed, Python is becoming an increasingly popular language to write books about. This is not the place to speculate as to why Python has attracted so much attention and not just from newcomers but also from those who are already experts in languages such as C++.

This is another book aimed at the person who wants to learn about Python. You do need some programming background because understanding

of programming fundamentals is implicit in the text. I suspect that the author does not recognise this and believes that the newcomer to programming could learn to programme (with Python) by studying this book. I guess someone with talent and the willingness to work hard at understanding might manage but this is not the book for the average newcomer.

It is a thorough introduction to the fundamentals of Python (and I will excuse the author a degree of language bigotry, because it is certainly milder than I have found from many authors – particularly those introducing Java) Of course, it is not an advanced text and only gives you an appreciation of the language whilst leaving you with a desire for some advanced texts. (I do not know why advanced texts for C++ seem to outnumber the sum total of advanced texts for all the other computer languages.)

I like the author's style of writing; it is relaxed and addressed directly to the individual reader. My Python is not good enough to make any in depth comments on the author's source code, however it appears to do what it is intended to do. In the case of languages such as Python that is probably a good enough test because unlike C, C++, Java etc. that have multiple implementations Python is effectively defined by its implementation. That is available free.

How does it compare with other books aimed at introducing Python? Well it largely depends on how you weight such things as writing style (easy going in this case), pace (relatively gentle), depth (stays away from the highly technical), interesting examples (well they aren't inspired). Another element that may matter to you is exercises. If you are studying by yourself the fact that this book does not have any exercises to help you consolidate your learning may be a big negative. On the other hand, if you were a teacher who liked to set his/her own exercises that might be an advantage.

Over all this is certainly a book to consider if you have decided to learn Python.

C# Primer: A Practical Approach by Stan Lippman (0 201 72955 5), Addison-Wesley, 392pp @ £34-99 (1.29)
reviewed by Huw Lloyd

The book's title draws an obvious parallel to Stanley Lippman's well known C++ primer; more importantly the title is also an accurate reflection of its content.

The first half of the book describes C# concepts. It provides a lucid description of all the C# keywords rationale and their use with appropriate examples. The reader is assumed to have rudimentary programming knowledge.

The second half provides some exposure to the .net programming environment: windows forms, web forms, the common language runtime (such as type reflection or meta data manipulation) and a 'Namespace' chapter that touches briefly upon many useful classes provided by .net.

The narration is clear and reads well, although many important aspects are written as footnotes and may be interpreted as 'small print'. The version used at the time of print appears to be V1.0.2914. All of the C# concepts discussed were compatible with my slightly more recent version.

This book will not suffice as a C# reference, for example, I would have preferred the index to have been more thorough. If you are performing some adventurous programming you will still need detailed reference material (e.g. MSDN).

Overall I found this book to be a good introduction into the C# language and the .net environment. The minimum knowledge required to implement frequently sought after implementations is provided with a sufficiently wide cross section of .net concepts that may be quickly adopted to achieve many complex programming goals. I am confident this book will satisfy the curiosity of many new C# programmers. Recommended.

Java etc.

The Java Developers Almanac 1.4 vol 2 by Patrick Chan (0 201 76810 0), Addison Wesley, 1026pp @ £22-99 (1.30)
reviewed by Francis Glassborow

This is the second volume of a book that I reviewed earlier. It is only just over a thousand pages. Most of these are solid pages of the public and protected interfaces of 22 packages that the book covers. Yes, there are some examples in the first third of the book, but they are the kind of example that only genuinely helps those who are fairly familiar with the material already.

The thing that depresses me about this book and its companion volume was that Java was originally described as a language that was substantially smaller than C++. That it is not the fault of the authors of this book. They have done the Java programming community a great service by placing so much at their fingertips.

The publishers should be thanked for keeping the price down (particularly as I suspect that in a couple of years time they will have to be replaced) and if you need this kind of information you now know where to get it.

Professional JMS Programming by various (1 861004 93 1), WROX*, 640pp @ £38-99 (1.28)
reviewed by Chris Czarnecki

Since JMS provisioning became mandatory in J2EE1.3 application servers and with EJB2.0 providing message driven beans, the role of JMS in enterprise applications has become more prominent.

This book provides an adequate reference on the features and facilities of JMS for those wishing to build robust, loosely coupled asynchronous Java applications. The book assumes knowledge of Java but to make the most of the text, readers really must also be familiar with JNDI, EJBs, Servlets/JSP and XML.

Less than a third of the book(200 pages) is dedicated to covering the JMS API and the JMS architecture. In these pages the reader is provided with a clear introduction to what can be achieved with the point-to-point messaging and also the 'publish and subscribe approach'. Version 1.0.2 of the API is covered, the latest being 1.1. In defence however, the book was published in 2000.

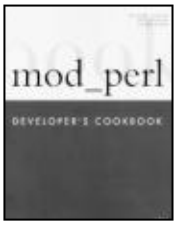
The age of the book really shows in the padding chapters (over 300 pages) with sections on using JMS in web application, EJBs, XML messaging and Mobile applications with all the message providers having updated their products from the ones used in the examples.

In summary, this book provides a good overview of the JMS API. However, it is spoilt with a large amount of padding that is now out of date and also in the context of the book mainly unnecessary. This, together with a price of £38.99 means there are far better and cheaper JMS texts available. Not recommended.

Introduction to Interactive Programming on the Internet by Craig Knuckles (0 471 38366 X), Wiley, 423pp @ £23-95 (2.49)
reviewed by Christopher Hill

[see web]

Other Languages



mod_perl Developer's Cookbook by Geoffrey Young et al. (0 672 32240 4), Sams, 645pp @ £28-99 (1.35)
reviewed by James Gordon

This is the sort of book I like, easy to read, lots of code examples, and even more description on how things work. It starts easy enough, installation of mod_perl and generation of code. Each section handles a single subject starting from the bottom and working up, again from the most widely used to the more esoteric. There's a nice list of 'almost' all of the constants and a list of resources at the back. The book is backed up with examples on their web site.

The layout is clean and easy to read with short code examples that are fully runable.

Modern Perl Programming by Michael Saltzman (0 13 008965 6), Prentice Hall, 340pp @ £31-99 (1.25)
reviewed by Pete Goodliffe

This is a good, considered introduction to Perl programming. Presented in a tutorial style, it has clearly been thought through carefully, well arranged and laid out sensitively.

It begins with a well-paced introduction to the language in chapters 1-7. This is not the most suitable introduction for a new programmer, but if you already know a programming language reasonably well it is pitched at the right level. There is no time wasted by labouring points, but everything is explained clearly and in sufficient detail.

In line with the book's title, Perl references and object-oriented programming in Perl are given good coverage. The OO section will only really make much sense to those who already understand the principles, but with this foundation it is a clear description.

Other 'advanced' topics covered include network programming, writing CGI scripts, GUI work, database interfacing and a crucial chapter on Perl debugging. None of these chapters will make you an expert in their field, but they are each a good introduction to get you up to speed.

The one real downside to this book is inherent

in Perl itself; the language is so large and has such a bewildering array of built-in functions and extension modules that it's difficult to cover them all in a tutorial-style book and this book doesn't. When you get into some serious Perl programming you'll want to stray outside of the scope of the text. That's inevitable.

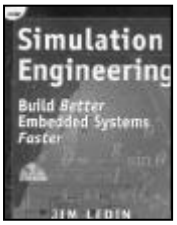
So we haven't replaced the Camel book here. The real shame, though, is that the book doesn't provide any further routes into information available. Clear pointers to other books, www.perl.com and www.perldoc.com would be invaluable and most definitely a description of perldoc (and how to use it) should be mandatory. It would also be helpful to have a section providing a quick overview of some of the commonly used Perl modules, so the reader could get a flavour of which wheels have already been invented.

That said, this is still a recommended book, take a look if you're about to step out in Perl development.

Perl & LWP by Sean Burke (0 596 00178 9), O'Reilly, 242pp @ £24-95 (1.40)
reviewed by Joe McCool
[see web]

SVG Programming: The Graphical Web by Kurt Cagle (1 59059 019 8), Apress, 586pp @ £35-50 (1.41)
reviewed by Roger Fretwell
[see web]

Embedded Programming



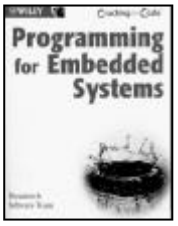
Simulation Engineering by Jim Ledin (1 57820 080 6), CMP, 302pp+CD @ £34-00 (1.32)
reviewed by James Amor

As anyone who has worked with them will know, embedded systems are a notoriously difficult beast to master. Simulation Engineering aims to assist the developer by introducing a set of engineering principles that assist the design, development and testing of the most complex embedded systems; in my opinion this aim is definitely realised.

All major aspects of simulation engineering are covered and Ledin does not shy away from any complex principles, providing comprehensive and relatively easy to understand explanations. A myriad of subject areas are covered including all areas of embedded simulation, data visualisation and analysis, verification and validation, software tools and management issues; to list the number of important principles introduced would far exceed the space I have to complete this review. The main criticism I have of this book is that many areas are extremely difficult to read, however this is mainly attributed to the complexity of the techniques being introduced; once these techniques are understood they should form an indispensable part of most simulation engineers knowledge base.

Upon initial assessment you may be put-off by the prevalence of mathematical formulae throughout the book, I would encourage you to persevere as the book introduces a number of important principles; however if you are not particularly mathematically minded, Simulation Engineering also provides evaluations and information on software tools that will perform these calculations for you and then explains how to interpret the results! Some of these tools are included on the bundled CD and walkthroughs of their use provided in the text.

In summary I would say that this is a book targeted primarily at reasonably experienced embedded developers who are interested in learning the principles of simulation engineering. I would not recommend this title to complete novices to the field, however anyone else with an interest should find this book useful - providing they have the perseverance necessary to read it!



Programming for Embedded Systems by Dreamtech Software team (0 7645 4954 5), Wiley, 533pp+CD @ £37-50 (1.33)
reviewed by Pete Goodliffe

This is an ambitious book that sadly misses the mark by some way. It is ambitious in that it appears to have a good coverage of different sorts of embedded technology, but this comes at the price of being overwhelming, superficial and too large.

It is split into two main sections. The first is an overview of embedded software development. The second (spanning pages 128-501, i.e. about 75% of the book) is a series of 'examples' of embedded systems, showcasing a broad range of technologies and problem domains.

The first section sets off to a dry and somewhat slow start but doesn't really dig in deep enough. I think this should be the most valuable part of the book (and could almost stand alone from the later example chapters). It doesn't, presumably relying on the second section to spell out a lot detail. The second section doesn't.

The example projects miss the mark somewhat and in fairness are sold too strongly - they are hardly professional-quality applications. Their presentation requires intense scrutiny to glean much information and the examples fail to highlight many of the particular problems and specific approaches to embedded programming.

Although the book skirts through many different environments, I was left wondering where the VxWorks or pSOS examples were. In fact a number of these examples are really just small desktop apps that are 'embedded' by running an embedded version of NT/XP.

The typesetting of the book is heavy and not clean to the eye. The code has been laid out without any consideration to the printed page. A little reformatting would have aided clarity greatly. The book's organisation is peculiar in places.

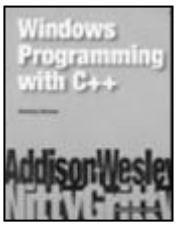
The authors often cave into industry hype and trot out a lot of dogma without explaining or thinking about what they're writing. They have a particular predilection for Java in this respect.

There is a supplied CD, containing material for both Linux and Windows, which is a nice touch. As with most such cover CDs it only appears to contain freely available material - Forte for Java, Java2 JDK 1.4, J2ME and RTLinux. It does contain the book's source code, which is more useful.

However, this book consistently doesn't address 'real' embedded issues. We don't need the incomprehensible source code for a naff MP3 player - the target audience should be able to read API docs and work out how to build one themselves. What's needed are descriptions of the real life balancing of performance issues, some examples of the difficulty of debugging in embedded environments, more on talking to real physical devices and something on the real concerns of working with RT OSes.

Overall, I wasn't that impressed. The book scratches many surfaces and presents a number of examples that don't really provide any usable, digestible keys into embedded development. It is reasonably expensive and reasonably large and would benefit from being a smaller, more targeted book, removing unnecessary waffle and duplication.

OS and GUI Programming



Windows Programming with C++ by Henning Hansen (0 201 75881 4), Addison-Wesley, 284pp @ £19.99 (1.50)
reviewed by David Nash

When I saw the title of this book, I wondered which class library it would use; the popular but oft-criticised MFC, the popular alternative WTL, the ATL, or even a home-grown system of classes.

I didn't even consider that the book would use no class library at all. In fact, with the exception of one chapter, no C++ facilities are used apart from declaring variables in the middle of blocks and the occasional memory allocation with new. The book teaches the standard Windows 32 C API.

The exception comes in chapter 20, which is dedicated to DirectX. In this chapter member functions are called on C++ objects. However no description of any of the classes being used is given, the objects are accessed via pointers obtained from standard C functions and the functions simply called via those pointers.

The very first sentence of the book is in a paragraph headed Requirements and reads, 'You need to have a good knowledge of the C++ language.' I would dispute that – you need to have a reasonable knowledge of the C language, together with a very basic knowledge of the C++ language. The book then goes on to describe how to use the Borland and Microsoft C++ compiler to compile a simple program – something I would have thought unnecessary in this book.

The method used by the book to teach virtually all the way through is to show a program listing then explain it. This is adequate in some cases, but in many I feel not enough explanation is given to what is being shown.

The book is split into three sections. The first is a tutorial of most of the features of the Win32 API, explained using examples as I said above. It covers many topics but I haven't attempted to determine whether that is exhaustive, but given that the section ends on page 146 I doubt it.

The second section is somewhat inexplicably called Take That! and is a reference for a selection of the Win32 API

chosen by the author to represent the 'fundamentals' plus the Graphics Device Interface (GDI) functions and a few others.

The third section contains miscellaneous extras including the chapter introducing DirectX mentioned earlier, a tiny (1 1/2 sparsely populated pages) chapter on Unicode, which simply lists the different versions of Windows and recommends whether you should use Unicode or not, one page attempting to describe COM (yes, really – I couldn't quite believe it) and only in the final chapter is the subject of compiling resource files mentioned.

If it weren't for the copyright date of 2002 (2001 in Germany) I would have thought that this was an old book hastily brought up to date by the use of the C++ features mentioned. For instance, when explaining the Windows message loop, is it relevant these days to say 'Applications in Windows are no longer run in the same way as DOS applications'?

The book is part of a series by Addison-Wesley called Nitty Gritty. It has been translated from German (well enough, I might add) and judging from the names of the authors of the other books in the series, those books will have been too. Unfortunately it doesn't come up to the standards we have come to expect from that publisher.

According to the back cover the series is supposed to teach the basic and most important facts. I would say that it succeeds in introducing the basic facts and for only £20 maybe we shouldn't expect more. However, I wouldn't recommend it and suggest that anyone looking to begin using the Win32 API starts with the classic Petzold rather than this book.

Designing from both sides of the Screen by Ellen Isaacs & Alan Walendowski (0 672 32151 3), New Riders, 336pp @ £34-99 (1.29)
reviewed by Christopher Hill

Think of the old butler in a black and white movie; speaking when spoken to; anticipating employer's needs; remembering how things were done last time; doing the job with out complaining. Over time, the employer learns how to make requests, spotting that she gets better results if she asks one way rather than another, picking up on his feedback. They develop a good relationship and learn to work together without noticing the interaction. They learn to collaborate.

The premise of this book is that there should be a relationship of collaboration between the user and the software. The software should allow the user to quickly get into a state of flow, where they can focus on the work that needs doing, without having to stop and persuade the software to do the task at hand. Isaacs proposes a number of principles grouped under two main headings.

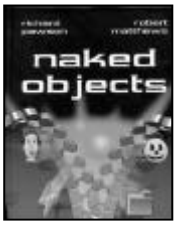
Don't Impose – make every click count, remember where they put things and remember what they told you.

Be helpful – use visual elements sparingly, make common tasks visible – hide infrequent ones, give feedback, follow conventions, solve problems – don't complain or pass the buck, be predictable and explain in plain language.

The book is split into two parts. The first part describes the principles with many examples of good and bad practice. The principles are easy to grasp and are full of those 'of course' moments. The second part takes the reader through the development of a real project, from talking with the customer to establish the requirement, through to laying out the user interface using storyboards. Only now are the programmers let loose to build the software and the architecture to support the user interface and we see how the initial problems are resolved and for use throughout the project, how to run Usage Studies.

A very easy to read book, yet packed with useful ideas for building collaborative software. Anyone designing and/or building any computer user interface should read the first part of the book and they will find they enjoyed the rest of the book as well. Highly recommended.

Other Programming



Naked Objects by Richard Pawson & Robert Matthews (0-470-84420-5), Wiley, 275pp @ £32-50 (1.69)
reviewed by Francis Glassborow

I recently got an email with the title of this book as the subject. I very nearly threw it away unopened but I recognised the sender's address and took a risk on opening it. I only mention this to highlight how careful people should be with the subject lines of emails, particularly when dealing with books whose titles have been chosen to make you look twice.

A couple of years ago I first came across 'Agile' methodologies and it took me a while to realise that 'agile' had been given a specific meaning in programming, in other words it had become jargon. It took me a little less time to discover that 'Naked Objects' wasn't just a title to grab the eye, though I wonder if everyone would feel comfortable to be seen reading a book with this title plastered in large letters across the cover. To clarify the issue let me quote a paragraph from the introduction:

> Naked Objects is an open-source Java-based framework designed specifically to encourage the creation of business systems from behaviourally-complete business objects. In fact, with the Naked Objects framework you have no alternative but to make your business objects behaviourally-complete. The reason is that the framework exposes your core business objects, such as Customer, Product and Order, directly to the user. All user actions consist of invoking methods directly upon those business objects, or sometimes upon the object's class. There are no scripts, no controllers, nor even any dialog boxes in between the user and the 'naked' objects. (Note: Wherever 'Naked Objects' is capitalized we are referring to the Java framework itself, and the term is singular. Where it is uncapitalized we are referring to business objects (plural) that are designed to work with the framework, and so are exposed directly to the user.)

This book is an excellent presentation of the subject and the publishers have taken care to use colour in various ways to help delineate various parts of the book. The book is hardcover and printed on quality stock. The contents start with a critical look at object-orientation before introducing naked objects. The author then introduces the Naked Objects framework before spending some time on the development process. It concludes with a brief chapter on extending Naked Objects.

The book also includes a number of case studies that help the reader put the theory into context.

There is good Internet support for the book and if you want to learn a little more before committing yourself to buying the book then a visit to www.nakedobjects.org would be advised. That is also the place to go if you are merely curious even though the subject is unlikely to have any application in your areas of expertise.

Overall this book is aimed at the subset of the Java Community that is concerned with business programming. If this includes you, then I think you should at least give the website a look and then buy the book if you want to pursue the subject.

Mastering Regular Expressions by Jeffrey E F Friedl - Second Edition (0 596 00289 0), O'Reilly, 460pp @ £28-50 (1.40)
reviewed by Joe McCool
[see web].

Pocket References

TOAD Pocket Reference for Oracle by Jim McDaniel et al (0 596 00337 4), O'Reilly, 120pp @ £8-95 (1.45)
reviewed by James Gordon
[see web]

C# Language Pocket Reference by Peter Drayton, Ben Albahari, and Ted Neward (0-596-00429-X), O'Reilly, 118pp @ £8.95 (1.45)
reviewed by Francis Glassborow
[see web]

JavaScript Pocket Reference by David Flanagan (0-596-00411-7), O'Reilly, 127pp @ £10-50 (1.42)
reviewed by Francis Glassborow
[see web]

Databases

Succeeding with Object Databases by Akmal Chaudhri & Roberto Zicari (0 471 38384 8), Wiley, 442pp @ £42-95 (1.16)
reviewed by James Gordon

This must be the hardest book I've ever tried to read and I've tried to read it a number of times before now. The book is a mix of high level history of OO databases; samples of UML, SQLJ, etc. and case studies. I think it is aimed at people with a lot of OO and relational database knowledge who want to turn their hands to OO databases.

There are chapters on different databases, i.e. Jasmine and Oracle and other chapters that show code for accessing an Oracle database using SQLJ and Java. The case studies range from genetics to maps to railways.

The book is hardback which makes it a nice looking book. It is very well written and if you are looking for a detailed overview of OO databases then this is a good book. If like me, you're more interested in designing and coding OO databases this book is a hard slog but worth it for the history.

Unix

Unix Weekend Crash Course by Arthur Griffith (0 7645 4927 8), Hungry Minds, 383pp+CD @ £18.99 (1.32) reviewed by Joe McCool

There's something quaint about this book. In the blurb about the author we are told 'Mr. Griffith was first introduced to UNIX in 1985, ...' Then 'Mr. Griffith spent several years ...' and 'Mr. Griffith's most recent books include ...'. Quaint.

Quaint too the suggestion that the reader grab a unix terminal over the weekend and work his way through Mr. Griffith's 30 learning sessions. If it is a weekend, chances are the reader will be using some Linux variant, not Unix at all. His default prompt will not, as Mr. Griffith claims, be "#" or "%". It is more likely to be something much more informative, but none the less confusing to the beginner. The bash shell will see to that. (In the section on scripting, bash doesn't get a mention! It is by far the most commonly used shell.)

Quaint too the suggestion that we mount the supplied cdrom via: mount /dev/cdrom /mnt/cdrom. Some unixes will whine that /dev/cdrom doesn't exist or /mnt. Quaint also the suggestions that we simply copy the voluminous software from the supplied CD and compile it up. SCO Unix, for example, doesn't even come with a compiler. I couldn't get the CD to work at all.

Books like this remind me of the 'Get Rich Quick' spam emails I am bombarded with. There is no free lunch. There is no quick way to learn unix. Such is the nature of the beast, much more than a weekend is needed to gain competence.

If you want to learn unix, borrow a Linux CD from a friend, join your local user group, subscribe to the newsgroups and jump in the deep end. It is worth the effort. The money you'll save on buying this book can be spent on a fishing rod and a box of worms. Worms are not so quaint.

Methodologies etc.

Advanced Use CASE Modeling by Franf Armour & Granville Miller (0 201 61592 4), Addison-Wesley, 425pp @ £26-95 (1.30) reviewed by Silvia de Beer

A well written, well structured informative book. The book simply fulfils its promises and explains use case modelling in all its facets. It would improve knowledge and confidence of use case developers. For small development projects, it normally suffices to read any short introduction to use case development, which can be found in many books. However, if you find it difficult to decide what to put in the use cases, to find a balance between details and readability, this book will help you reflect on your own use case development process. Also, for those involved in a larger development project, it can be productive to have a better background understanding of the purpose of use cases.

The main purpose of use case development is to specify the requirements and besides that one must not forget the non-functional requirements. Use case modelling is not the same as designing the architecture of a system and a pitfall is to use the functional breakdown directly in an object-oriented implementation.

It surprised me a bit when reading this book, that there is actually very little about the notation of use cases. I remember very well my first use case development experiences, where there was discussion about the direction of arrows. Of course, the notation is explained in this book, but there is actually very little to it. More importance is put onto the fact that use cases should convey the requirements and that they should be verifiable by the client. The book describes well how to start use case modelling and which iterations could be taken. All explanations are correctly supported by diagrams and tables and supported by a case study throughout the book.

One of the confusing points about use case development for me was always the use of <<uses>> and <<extend>> relationships. The book is based on UML 1.3, which removed the <<uses>> relationship from the previous standard and added the <<include>> relationship. In one of the chapters the correct use and understanding of those relationships is clearly explained, so no need anymore for confusion.

Facts and Fallacies of Software Engineering by Robert L. Glass (0-321-11742-5), Addison-Wesley, 195pp @ £22-99 (1.30) reviewed by Francis Glassborow

This is the latest title from a well-known author in this subject area. Alan Davis has this to say in his brief Foreword:

> Bob has had a history of providing us with short treatises on the many software disasters that have occurred over the years. I have been waiting for him to distil the common elements from these disasters so that we can benefit more easily from his many experiences. The 55 facts that Bob Glass discusses in this wonderful book are not just conjectures on his part. They are exactly what I have been waiting for: the wisdom gained by the author by examining in detail the hundreds of cases he has written about in the past.

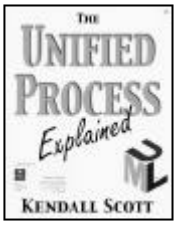
In the first four chapters (About Management, About the Life Cycle, About Quality and About Research) Robert Glass presents 55 'facts'. Each is presented with a single sentence introduction (e.g. fact 7: Software developers talk a lot about tools, but seldom use them.) Under that heading you will find four sections: Discussion, Controversy, Sources and References.

The final three chapters (About Management, About the Life Cycle, and About Education) present 10 fallacies (for example, fallacy 3: Programming can and should be egoless. and fallacy 10: You teach people how to program by showing them how to write programs.) are presented in a similar format.

The author has a comfortable writing style that makes it easy to read and understand. Of course, the main reason for writing a book such as this one is that much of the content will be considered controversial by many of its potential readers. I never advocate thoughtless reading of technical books, and that includes thoughtless rejection as well as acceptance.

The biggest problem is that different items in this book are addressed to different participants. By that I mean that the people who have the power to respond to them vary from managers, through system architects and programmers to educators. This means that the individual reader is only in a position to respond to parts of the book. I hope that many of you will read the whole of this book but not then proceed on the basis that you are OK and what really needs fixing is the work of other types of participant.

This is a book that deserves to be widely read by participants in software development, and then discussed, understood and acted upon.



The Unified Process Explained by Kendall Scott (0 201 74204 7), Addison-Wesley, 185pp @ £26-99 (1.30) reviewed by Paul S Usowicz

As soon as I receive a book I have a quick flick through just to see what I can expect. It soon became apparent that had I flicked through this book in a bookstore I would not have purchased it. Not because the book is no good but due to its irrelevance to the way I currently develop software.

The book is, in fact, extremely well written and very clear with easy to follow guidelines and I thoroughly enjoyed reading it. The whole unified process is explained from start to finish with pointers on what to look out for and some useful examples. I found the sections on testing to be extremely useful and full of common sense.

So why is this book not for me? The whole unified process is geared towards large software developments. You need architects, testers, coders, etc. Although these can be the same person the overhead of the process is probably too great for my one man, one-month stand alone executables. If you run a large software department, however, I can see great benefits in adopting the process. I will hold on to this book, however, as I will definitely refer back to it if my projects increase in size.

My one bugbear with the book is that it constantly refers to one of the author's previous books 'UML Explained'. I don't buy a book to be told I really should buy another one as well. Some of the diagrams are explained fully in the other book and only referenced in this book. If the explanation is necessary for a full understanding then it should be reproduced in full.

Non-Programming

Hack Attacks Revealed 2ed by John Chirillo ((0-471-23282-3), Wiley, 913pp+CD @£44-50 (1.35) reviewed by Francis Glassborow

[see web]

Windows XP Annoyances by David A. Karp (0-596-00416-8), O'Reilly, 565pp @ £20-95 (1.43) reviewed by Francis Glassborow

[see web]

Unicode Demystified by Richard Gillam (0-201-70052-2), Addison Wesley, 852pp @ £37-99 (1.32) reviewed by Francis Glassborow

[see web]