ISSN 1354-3172

Overload

Journal of the ACCU C++ Special Interest Group

Issue 14

June 1996

Editorial:
Sean A. Corfield
13 Derwent Close
Cove
Farnborough
Hants
GU14 0JT
overload@corf.demon.co.uk

Subscriptions:
Barry Dorrans
2 Gladstone Avenue
Chester
Cheshire
CH1 4JU
barryd@phonelink.com

£3.50

Contents

Junk Mail

I believe that Capita Recruitment Services have misrepresented their relationship with ACCU in a recent mailing. They purchased a set of mailing labels from us for C++ SIG members who had not restricted use of their contact details for this purpose. What Capita seem to have done is to use those labels to produce mail-merging data. This is strictly contrary to the terms under which we provide such label sets. ACCU does not provide details of members to other organisations except in the strictly limited form of sets of one time mailing labels. These are always identifiable because of the style of printing and layout.

I hope that members have not been too inconvenienced by this junk mail. I particularly hope that it will not cause anyone to restrict use of their address in future because the sale of mailing labels produces finance that helps us keep membership fees as low as possible.

Francis Glassborow
francis@robinton.demon.co.uk

Editorial

Submissions

For this issue, I'd just like to say a couple of things about submissions for *Overload*. Firstly, there simply aren't enough of them! This issue does see some new contributors whose presence is very welcome. You don't have to be a great writer and you don't have to be a C++ expert to submit articles for *Overload*.

Secondly, preferred formats. I really do prefer plain text. It makes my life easier. If you must send me formatted material because you want me to see how it should be laid out, RTF is a good bet. I can accept Word format but the translators between different versions on different platforms can cause subtle bugs and formatting problems (one issue was delayed by a week as I tried to excise a particularly nasty pagination gremlin introduced by the Word 2.0 to Word 6.0 convertor!).

Thirdly, email. A couple of folks have sent printed submissions and these have a habit of getting lost amongst my other paperwork. If you're on CompuServe and want to send me a binary attachment (e.g., for a Word file), send it to my CompuServe account 101554,1127 but otherwise, send email to the address below. If you don't have email and want to send me an article, please enclose a disk with the soft copy on it. I really am quite allergic to paper and I'm extremely likely to lose it!

ACCU and the Internet

I get several requests each month to provide all the useful email and web addresses for ACCU related sources. Due to problems with Demon and some ACCU hardware, the email and web forwarding has been a little unreliable recently but these problems are being sorted out and from the next issue onwards I will print a list of useful email and web addresses in each issue.

Sean A. Corfield
overload@corf.demon.co.uk

Software Development in C++

This section contains articles relating to software development in C++ in general terms: development tools, the software process and discussions about the good, the bad and the ugly in C++.

Francis Glassborow takes a close look at our terminology for values, Keith Derrick begins a series looking at implementing well-known patterns and Dave Durbin provides more information about Java after The Harpist's introduction in *Overload 11*.

Concerning values, left, right and converted

by Francis Glassborow

What sort of things are: `12`, `2.3`, '`A`', "`Help`"?

Well in programming context they are all literals. In C++ they might be described as: an **int**, a **float**, a **char** and a string. Using such terminology serves to confuse many. Let me take a few further examples.

What sort of things are returned by: **int** *fn***()**, **double** *sqrt***()**, **char*** *xyz***()**? The simple answer is 'values', the more complicated answer is 'r-values' but more of that in a moment.

Given:

```
int i;
float f;
double x;
```

What are *i*, *f* and *x*? The simple answer is: 'an **int**', 'a **float**' and 'a **double**'.

The problem I am getting at is that we use type to describe two related but distinct concepts, storage and value (container and contents). Most of the time we are not even aware that we are using type names for two different things. C introduced (more precisely, it redefined) two words, 'lvalue' and 'rvalue' to denote the two ways in which a type can be used. In simple terms an lvalue is a container and an rvalue is contents. Literals and return values are necessarily rvalues, contents not containers. Variables are something else as they can be both an rvalue and an lvalue depending on the context in which they are used. Mostly this distinction is clear in context and we are not confused. However confusion arises in two places: iterators and conversions.

Iterators

The Harpist suggested that you think of an iterator as a generalised pointer. Fine as far as it goes but most programmers get confused by pointers. So let me spend a moment examining the meaning of a pointer in C/C++.

Just as for any other type we can have both pointer lvalues and pointer rvalues. Just in every other case, a pointer lvalue is storage for a pointer rvalue. What confuses is that the term 'pointer' suggests that in some way the object points to something. While sensible, it causes confusion. A pointer rvalue does indeed point to something, and we would more naturally call that an address. A pointer lvalue does not point to anything, it is storage for a pointer rvalue (address).

Think of the number of books that tell you that, if:

```
int* ip;
```

then *ip* is a pointer to **int**. That is, indeed, its type but it is not what it does. As an rvalue its contents point to an **int** storage location. That is, its contents are the address of the storage for an **int**. We know that is the case because we confidently write:

```
int i;
ip = &i;
```

And call **&** an 'address of' operator not a 'get pointer' operator. I think it is much easier to talk about pointer variables as pointers and their contents as addresses. My response to 'what does **char*** *fn***()**' return?' is 'address of a **char**.' Similarly, 'If **int** *list***[10]**, what sort of thing is *list*?' meets the response 'address of **int'**. That is why we need a pointer parameter to receive an array argument, the array is passed as an address and pointers are the things that contain addresses.

Please think about this. The rvalue of a pointer variable is an address, the lvalue of a pointer variable is the location where an appropriate address can be stored.

By the way, a reference parameter is something quite different. It does not denote any form of new storage. It provides an alternative identifier that can be bound to existing storage. The nearest analogy that I can come up with is that it is a little like:

```
extern int i;
```

Which declares *i* to be the name of storage for an **int** that is defined somewhere else. A reference declares an identifier to be an alternative name of storage defined by another name elsewhere.

I think C++ made the already confusing pointer terminology worse by talking about iterators. The first thing that springs to most people's minds when the term iterator occurs is that it must be something that allows them to iterate over a collection. It is only those that have a secure grasp of type terminology that recognise that, if it is a type, it will occur in two flavours, lvalue and rvalue. A variable of an iterator type provides storage for something that can be progressively modified to iterate over a collection of objects. But in the terminology of C++, that something is also called an iterator (meaning an iterator rvalue).

When I first came across the statement that the *begin***()** and *end***()** member functions of the STL container classes returned iterators I was completely bewildered. It took me quite a long time to understand that what was meant was that these functions returned iterator values that acted as starting and finishing values. For example:

```
int ray[100];
for (int* iter=ray; iter<ray+100;
iter++)
{
 // do something
}
```

is a simple C-style container with an 'iterator'. In standard terminology *iter* is a pointer to **int** that is initialised with a pointer value (*ray*), then incremented while the value in *iter* is less than another pointer value (*ray* **+ 100**). I find it easier to read that as: *iter* is a pointer to **int** that is initialised with the base address of *ray*, and stepped through in **sizeof(int)** steps while the address in *iter* is less than the address of one beyond the end of *ray*.

Compare this with:

```
vector<int> vt(100);
for (vector<int>::iterator iter =
     vt.begin(); iter!=vt.end(); iter++)
{
  // do something
}
```

Even if I know that the iterator type for *vector*s in my implementation is a plain C-style pointer, I would be wrong to use that information because that is an implementation detail. It is quite possible that it has been replaced by some other iterator, such as some kind of smart pointer. Note that the comparison has changed from 'less than' to

'not equal to'. That is important because there is no guarantee that 'less than' will be defined for all iterators, however we do require that **operator==** and **operator!=** are defined for an iterator. Note that in the case of smart pointers, these will be defined because there will be a chain of conversions that terminate with raw pointers. The problem with 'less than' in these circumstances is that there is no guarantee that all the converted intermediate values of the iterator as it steps from *begin***()** to *end***()** will actually be less than *end***()**. It will work for vector because of the requirement for contiguous storage but not for other STL containers.

I think that the concept of an iterator as some form of generalised pointer is fine, as long as you are clear about the terminology that uses 'pointer' to refer to both the contents and the container. Functions such as *begin***()** and *end***()** return iterator values that can be stored in iterator objects.

STL specifies five groups of iterators: Input, Output, Forward, Bidirectional, Random.

After consulting Francis, I have decided to publish the following explanations unaltered. In fact, they contain several incorrect assumptions about the iterator categories but, as I hope I illustrated in "You can't get there from here" in Overload 13, iterators are subtle and the requirements on them are complex and easily misunderstood. I shall run an article in Overload 15 on this subject – Ed.

Input/Output:
: These can be dereferenced and incremented. The process of dereferencing is sequential in that each time you use it the iterator will be incremented so you can only read (Input) or write (Output) to a location once through a specific Input/Output iterator. This may seem restrictive, but it makes perfectly good sense in context.

Forward: Like the previous case, except that read and write do not increment the iterator. That means that a single object can be used repeatedly until and increment operator moves you on. Classic single linked lists (where each node is only linked to the next one) are candidates for forward iterators because you can easily move on, but only serious contortions allow you to move back.

Bidirectional:
: Like the forward iterator except that the decrement operators are also supported. Because STL linked-lists are doubly linked lists (ones where each node is linked both to the previous and the next node) they are suitable candidates for bidirectional iterators.

Random: These support all the operators that you would associate with raw pointers, including indexing. In STL, the *vector* container is a candidate for random iterators. If you want the *n*th element of a *vector v* you can write ***v*.*begin*()[*n*-1]**. This will also work for *deque* containers, though the iterator certainly will not be a raw pointer.

The average applications programmer will not generally be creating her own iterator types though she may do so incidentally when she creates special smart pointers for debugging tasks. The task of creating new iterator types is in the domain of the class implementor. If she gets that job done correctly it will be an implementation detail that will be transparent to the application programmer who is a client of that class.

Applications programmers need to know what to expect when a class says it has a *X* type iterator so that they know how it can be used. Much of the problem with current C++ programming is that there is often no division between the applications domain and the implementation domain. Even if you are forced to be both, you should have a clear understanding as to which role you are occupying at any given moment.

Conversions

I was recently profoundly shocked by a programmer emailing me about a piece of my code on the grounds that there was no default **operator<<(***ostream***&, const** *T***&)** defined where *T* was an **enum** so code such as;

```
#include <iostream.h>
enum X {zero, one, two};
int main(){
       X x=zero;
       cout<<x;
       return;
}
```

should not compile. This, along with other recent correspondence made me realise just how tenuous a grasp some programmers have of C++

conversions. So I thought that it might be worth writing a little on the subject (you can all pick holes in it if you like).

Basically there are three main groups of conversions, implicit (compiler can use without programmer action), explicit (can be used via a **static_cast**<>) and forced (requires a **reinterpret_cast**<>). There are a number of rules that you need to know.

First, implicit conversions break into two groups. There are standard conversions and user defined conversions. The standard conversions include all the inter-conversion between built-in types including conversion from a *T****** (where *T* is any data type) to **void*** but not the reverse. In addition an **enum** type can be converted to an **int**, a derived type can be converted to a base type. If you know of any others please write in.

The user defined conversions are all constructors that can take a single argument and haven't been marked as **explicit** (only possible in the most up-to-date compilers) together with all conversions provided by **operator** *T***()** where *T* is some type. It is because these provide implicit conversions that programmers should be particularly careful about providing such conversion operators.

Implicit conversions are not just single step conversions, the compiler can use any sequence of conversions that consist of standard conversions and not more than one user defined conversion.

For example:

```
enum X;
class T {
       // what ever
public:
       operator X();
       T(X);
       // rest of definition
};
```

Empowers the compiler to use a *T* object wherever any built-in numerical type is required. The constructor does not allow construction from any numerical type because there is no standard conversion from a numerical type to an **enum**.

Next, explicit conversions. The reverse of any standard conversion is available as an explicit conversion, this includes standard conversion sequences. This means that you can, if you insist, cast a **float** to an **enum** type; the compiler cannot do it off its own bat, but you can if you wish.

In addition, any single argument constructor can be used for explicit conversion together with all implicit conversions. Again, any conversion sequence can contain at most one user defined conversion – if you need more than one then you must make the extra ones explicit as well. In other words each explicit conversion down the chain must be made visible.

The correct cast for an explicit conversion is a **static_cast**<>. If you want to change **const**/**volatile** qualification as well then you will need a separate cast to handle that (**const_cast**<>).

Note that conversions are carried out on rvalues even if the result is stored either permanently or temporarily in an object (lvalue), they do not and cannot change the original.

What about that last group of conversions, 'forced'. You may know that a specific bit-pattern representing a value of type *T* can also represent a value of type *Q*. Under such circumstances you can instruct the compiler to use the bit pattern of an object of type *T* as the bit pattern of an object of type *Q*. This is done with a **reinterpret_cast**<>. I have over-simplified this because all that **reinterpret_cast**<> requires is that the relevant bit patterns are interchangeable in the sense that **reinterpret_cast**<> back to the original type will restore the original value.

I find it difficult to find good examples for using **reinterpret_cast**<>, and I am sure that many others also find it difficult. I would welcome reading an article about useful uses of this cast. If you know any please share them with the rest of us.

WARNING

Throughout this article I have taken liberties with terminology and much of it would cause gagging among my fellow standard panel and committee members. I have tried to write in terms that give the ordinary working programmer a fair chance of gaining some insight. Of course my understanding may itself be faulty but then those that write in to correct it will be doing all of us a service.

Francis Glassborow
francis@robinton.demon.co.uk

Real world patterns

by Keith Derrick

Introduction

Last year, a quiet, unassuming book was published by Addison-Wesley with the simple title "Design Patterns". It was soon discovered by the industry and rapidly became a standard reference for class designers and implementers alike.

The authors (Gamma, Helm, et al) had spent considerable time talking to heavy duty users of C++ - both designers and implementers - gradually building a collection of popular approaches to resolving common design problems. Then came the stroke of genius: each approach was generalised; beaten into shape; named; and finally described in a standard format. This produced a cook book of ideas which could be used by class designers to describe design attributes of a class in a standard way.

For example, say a class has a cardinality of 1 - i.e., there must only ever be one instance of the class in existence at any time. The designer can simply state that the class should be implemented as a *Singleton*. The *Singleton* pattern will be understood by both designers and implementers, so the designer can concentrate on the class-specific aspects of the design.

You will have seen references to some of the patterns in other articles in recent issues of *Overload*, and the C++ press in general. I predict this will become more and more common, which is of course the aim of the book's authors - if only in that respect, they have been hugely successful.

Although I no longer consider myself a complete novice in C++, I am far from being an expert. This book has allowed my own attempts at class design to take a quantum leap forward in both quality and success. Since buying the book I have been spending much of my spare time evaluating the various patterns and now regularly incorporate some of them into applications being written for my clients.

In this series of articles, I hope to share the knowledge and understanding I have gained with those who are a little behind me in the learning curve. Hopefully, I will also learn some valuable lessons from the comments of readers - experts and novices alike.

Given that many compilers still do not provide full implementations of language features such as exceptions and templates, I will try to avoid relying on these. You should be able to try these patterns out using a compiler as old as Turbo C++ V2, or Visual C++ 1.0.

Now on with the first pattern. True to sod's law, considering the last paragraph, this one really cries out for implementation as a template!

Proxy

One of the more common mistakes made by C++ programmers, expert and novice alike, is failing to release all dynamically created objects - otherwise known as the memory leak. The following code fragment shows what I mean:

```
bool DoSomething(char* filename)
{
  MyClass* object = new
MyClass(filename);
  ...
  if (errorOccurred)
    return false;
  ...
  delete object;
  return true;
 };
```

Something goes wrong, and 'object' is lost forever. We've all done it: and if you haven't yet, you probably will soon - unless you learn the lessons of the *Proxy* pattern. The approach is simple in concept - let someone else take the strain of remembering to clean up the mess !

In addition to defining *MyClass*, also define *MyClass_Proxy* as follows:

```
class MyClass_Proxy
{
public:
  MyClass_Proxy(MyClass* obj = 0)
  : theObject(obj) { }
  ~MyClass_Proxy()
  { delete theObject; }
  MyClass* operator ->() const
  { return theObject; }
private:
  MyClass* theObject;
};
```

change the allocating line in our function to read:

```
MyClass_Proxy object(new
MyClass(filename));
```

and remove the **delete** *object* near the end of the function. The function should re-compile cleanly, and there have been no changes to its logic so it should still work the same. But the memory leak has simply disappeared!

The trick lies in the change of type for *object* from a pointer to an instance of one class, into a

concrete instance of another. Now, when *DoSomething* exits - whether via an exception, or an early return - *object*s destructor will always be called and will delete the dynamically created *MyClass* instance.

Cleaning up classes

A common technique with classes is to have a pointer data member which is initialised sometime during an object's lifetime. The destructor ensures the resource is deleted at the end of it's lifetime. The following example shows what I mean:

```
class MyClass
{
public:
  MyClass() : m_pObject(0) { }
  virtual ~MyClass()
  { delete m_pObject; }
  void AnOperation ()
  { m_pObject = new AnObject; }
private:
  AnObject* m_pObject;
};
```

The use of a proxy class for *AnObject* streamlines the implementation and specification of the client class by "hiding" some of the implementation aspects of the design decision to use a pointer.

```
class MyClass
{
public:
  ...
  void AnOperation()
  { m_object = new AnObject; }
  ...
private:
  AnObjectProxy m_object;
};
```

You can't take it with you

Another source of memory leaks are full lifetime instances which are created when they are first needed, and then used throughout the life of this program's execution. Many programmers either forget to, or simply choose not to, delete these instances. Some operating systems and compilers will clean up behind you - others will crash!

By declaring a *Proxy* variable at file scope, you guarantee its destructor will be called before the program finally exits. You also avoid that unsightly mess at the end of *main***()** where you have a list of delete statements.

Refining the Proxy

By now some of you will be feeling a little worried about the proxy class I presented above. Now we need to make it a little more robust.

First, given that a *Proxy* object will always point to a dynamic instance of a class, it would make no sense to try creating a dynamic *Proxy*. To avoid this, we want to enforce the condition that *Proxy* objects may only be defined as file scope, automatic, or class member variables. We do so by overriding the **new** and **delete** operators for the class, and declaring them as **private**. Now the only place they can be used is within a member function and we're not going to do that.

Next, we do a similar thing for the copy constructor and assignment operators. The last thing we want is two *Proxy* objects pointing to the same dynamic instance!

Most of you will want to be able to use the *Proxy* as an lvalue to initialise it as in

```
Proxy anObject = ...;
...
anObject = new Object;
```

To achieve this we provide an assignment operator. I choose to simply delete any current object to which the *Proxy* has a pointer; alternatively, this could be considered worthy of an exception!

And of course, you will want to use it in place of a normal pointer, so we overload the -> operator to return a pointer-to-object value. Overloading the -> operator instead of providing a conversion operator has the added advantage that **delete** *proxy* will not delete the dynamic object. Any cases where a pointer-to-object really is needed can be catered for by an explicit accessor member.

Or writing proxy.operator->() – Ed.

We now have something like the following

```
class Proxy
{
public:
  Proxy(const Object* obj = 0)
  : theObject(obj) { }
  ~Proxy()
  { delete theObject; }
  Proxy& operator=(const Object* obj)
  {
    delete theObject;
    theObject = obj;
    return *this;
  }
  Object* operator->() const
  { return theObject; }
private:
  Object* theObject;
  //
  // Declare the following as private so
  // they are disabled. No
implementation
  // should be provided. That way if it
  // gets past the compiler, then the
  // linker should still complain
```

```
  //
  void* operator new(size_t);
  void operator delete(void*);
  Proxy(const Proxy&);
  Proxy& operator=(const Proxy&);
};
```

One final note, the following shows a problem with our *Proxy* implementation. I'll leave it to you to find it suggest possible solutions - I know of only one.

```
MyClassProxy proxy = new MyClass[10];
```

Keith Derrick
kderrick@cix.compulink.co.uk

More on Java

by Dave Durbin

This was a letter forwarded to me by Francis but I felt it contained such a lot of useful information about Java that it warranted an article of its own! – Ed.

First of all, allow me to introduce myself. My name is Dave Durbin and I am a recent subscriber to the ACCU.

My background is as a software developer specialising in client/server and object oriented solutions with particular regard to distributed object based environments (including the Internet). I have several years experience with C and somewhat less with C++ (I have recently reached the level of writing genuine OO code as opposed to 'C with objects' style programming). I am employed by IBM and work in Edinburgh.

I'd like to take this opportunity to thankyou for your excellent work with *Overload*. I have found it particularly helpful in explaining and exploring some of the idioms employed by experienced programmers which are an essential part of any C++ programmer's repertoire.

That said, the main purpose in my writing is to raise a few comments regarding The Harpist's article on Java in issue 11 of *Overload*. Like anyone else, my experience with Java is small (after all, implementations of the language have only been available for a few months) but I have been working hard to familiarise myself with both the Java language and the architecture of the Java virtual machine on which Java bytecode executes. This letter is intended to add a little additional information to that provided by The Harpist. Please note that this is a very high level discussion document and as such I may oversimplify somewhat. Please let me know if you require clarification or more in depth description of any points which I make.

At the time of writing, Java v1.0 is now available across several platforms specifically inclding OS/2, AIX, Windows (NT and 95), Mac OS 7.5 and Solaris.

I always find it unnerving to see computer languages referred to by version numbers ("Java v1.0"). Once a standard is available of course, the document itself can be used, e.g., C89 or ISO C, but C++ was plagued by the gross misunderstanding that it was somehow tied to the version of Cfront, the archetypal C++ compiler – Ed.

There are three main technologies which are being generically referred to as Java. These are the Java virtual machine, the Java language and the HotJava web browser.

The JVM is specified by Sun (who have now implemented a version as an OS on silicon which they are marketing as a 'dumb' Internet terminal). It is a stack based architecture which exectues Java bytecode (the output of Java compilers). This is where the main strength of Java lies. Once code is compiled, it can be executed unchanged on any implementation of the JVM. This is true binary level compatibility, something which we have as yet not seen and which is fundamental to the open distributed computing model presented by the Internet.

Binary level compatibility has been achieved before with virtual machines. I would point interested readers at both UCSD Pascal (whose language extensions Borland successfully appropriated for Turbo Pascal) and Microfocus's COBOL compiler. Both of these can generate bytecode which is absolutely portable across all platforms that support an interpreter (the "virtual machine") – Ed.

The HotJava browser is a web browser, written in Java and built around the Java virtual machine. This means that it functions both as a conventional browser and has the ability to load and execute Java code (applets) across the Internet and execute them locally. This effectively means that the browser is infinitely extensible in terms of both protocol support and of function. Other browsers supporting the ability to execute Java applets include IBM's Web Explorer (v1.03) and Netscape Navigator (v2.x).

Mac users should note that Netscape Navigator v2.x does not support Java although v3.0 does. Hurray! I must confess to disappointment at the number of Java applets that fail on the Mac due to null pointer dereference. – Ed.

This obviously raises questions about security which The Harpist mentions. Who is going to trust applets imported over the network? They could contain fatal bugs or worse still, intentionally hostile code.

Fortunately the developers of the JVM had considered this. All code loaded is subject to validation. This involves verifying that the Java bytecode is valid and does not attempt to perfom illegal operations (overflowing or underflowing the stack, accessing variables as pointers, performing invalid casting operations etc). This is necessary as although the official Java compiler generates 'clean' bytecode it is possible that an attacker might tamper with the bytecode or that a damaged or buggy compiler may generate invalid bytecode.

Additionally, bytecode which is loaded across the network is subject to even more stringent security - it is not allowed to read or write to the local file system, nor to make any form of enquiry regarding its host environment. Finally, although Java supports a rich set of TCP/IP communications classes, code loaded across a network may communicate only with the server from which the code itself came or the server from which the HTML page in which it was embedded came. These features make running code from across a network a very safe proposition indeed.

(As an aside to the above, two flaws with Java security have been recently uncovered. One has already been patched, the other will be fixed very soon)

The basics of the Java language are fairly well documented by The Harpist. I do however feel that it is a mistake to compare it with C++. Java is syntactically similar to C++ but in terms of semantics has more in common with Smalltalk. It is far better to approach it as an entirely new language.

C/C++ features not supported by Java

These include: preprocessor, templates, operator overloading, pointers, memory management, multiple inheritance, destructors, goto, typedefs, macros, structs, enums, functions, header files.

Non-C++ features available within Java

These include: automatic (asynchronous) garbage collection, built in support for arrays and strings, interfaces (in its OO meaning), RTTI, windowing toolkit, TCP/IP networking classes, thread management.

*Being pedantic, I would argue that C++ does have RTTI and the standard library contains both arrays (**vector**<>) and strings – Ed.*

As mentioned above, one use for Java is to write applets which can be distributed via the Internet. It can also be used to write significant standalone applications with no network connectivity. In fact the Java compiler, debugger and the HotJava browser are all written in Java.

It is possible to link existing C (and via the usual convolutions C++) DLLs as 'native methods' within Java code. There are a lot of caveats regarding how (and indeed if) you should do this, mostly relating to portability issues and garbage collecting (how does the JVM know if an object created on the heap in C code is referred to from within the Java code?)

Finally (and at the risk of making The Harpist 'profoundly unhappy') I am not sure that I concur fully with his views that Java will not replace C++.

I agree with his summary of the strengths and weaknesses of Java and the likely application of the language but I think that he has seriously underestimated the percentage of applications which will be expected to operate as small interacting packages, over a LAN or the Internet and to be fully portable (at a binary level) across client platforms.

The Internet is just beginning to penetrate the consumer marketplace with companies like Philips, Sun, Panasonic, Sony etc. vying to get the first Internet 'set top boxes' onto the shelves and into peoples houses. This will inevitably lead to a standardisation on an abstract hardware model for these 'net connected consumer devices.

The requirements for security and interaction between these devices will be such that a language like Java will be essential for anyone aiming to provide software which will execute on these platforms. (NB the abilities of the high level language **are** constrained by the underlying

metal, it is not AFAIK possible to implement a C++ compiler which generates Java bytecode).

Mainly because C++ language features are effectively a superset of Java. Implementing a C compiler for the UCSD p-code system also requires minor extensions to the bytecode definition (if my memory serves me correctly) – Ed.

Anyone who intends to be running more than a local business after the next 3 years had better be servicing their clients via the Internet. It will be simply impossible (and certainly highly undesirable) to provide servers which can handle the number of potential simultaneous connections from external clients and so I believe that a move to offloading function from servers to clients is inevitable. Java or more likely one of it's descendents will be the language which fills that role.

The changes I describe above are not purely speculation, they can be observed happening on a small scale today. Equally clearly, they will not occur over night. What I do believe is that a Java-like (binary portable, simple, small, secure) language coupled with a massive market for it will marginalise languages like C++ at some point in the next few years (certainly within 5 years).

Of course C++ will not disappear and there will still be areas where it is preferred (much as assembly code is still used today where platform independence needs to be sacrificed for performance) but I suspect that it will be used only out of necessity.

I will be happy to answer further queries on Java (the language or the technology) to the best of my limited ability. I would however refer interested readers to the following sites :

`http://www.javasoft.com`
: The official Java/HotJava website with language definition, API documentation, VM specification etc.

`http://www.hursley.ibm.com`
: Download Java for AIX and OS/2. Also links to other Java sites

`http://www.netscape.com`
: Java enabled versions of Netscape Navigator for most platforms

I would also recommend following the `comp.lang.java` Usenet news group.

This is very high volume! – Ed.

A good introductory text is:

Hooked on Java – van Hoff, Shaio and Stabuck (members of the original Sun Java team), ISBN 0-201-48837-X

Dave Durbin
100102.2062@compuserve.com (preferred)
durbind@ibm.net

NB All of the views expressed above are my own and do not necessarily coincide with those of my employer.

I'm grateful to Dave for providing more information on Java. As before, I would be happy to run a regular series of articles on Java if there is sufficient interest (and sufficient contributors!). I have interspersed some comments above where I felt appropriate but I have one comment I've saved for the end. There's no doubt that Java is flavour of the month right now and a very interesting and practical language. However, it's dynamic nature means that it will forever be excluded from a large segment of the computing world: embedded systems and, probably, scientific computing. Both of these areas are showing a trend from existing favoured languages (typically C and FORTRAN respectively) towards C++ which still provides the benefits that both those arenas require, namely performance and the ability to control and predict runtime response (by controlling or restricting dynamic memory allocation and other runtime- and resource-critical issues). Ed.

The Draft International C++ Standard

This section contains articles that relate specifically to the standardisation of C++. If you have a proposal or criticism that you would like to air publicly, this is where to send it!

Overload 15 will see a report on the Stockholm meeting of WG21/X3J16 which should also see the production of the second Committee Draft which will signal the second ANSI public review. The most controversial issue yet to be decided is the template compilation model. If the Stockholm meeting confirms the

Santa Cruz resolution (see *Overload 13*), I will write an article explaining the decision, the background to the decision and its impact on your code.

C++ Techniques

This section will look at specific C++ programming techniques, useful classes and problems (and, hopefully, solutions) that developers encounter.

The Harpist continues his series on templates in the standard library, Jon Jagger looks at some of the issues involved with encapsulating time and provides some 'random' musings on function-like objects and Francis asks us to question a popular idiom.

The Standard Template Library – first steps auto_ptr

by The Harpist

Now, before all the experts shout at me, I know that *auto_ptr* is not actually part of the STL, but it is part of the Standard C++ Library and it is a template class. In my opinion, like the sequence containers that I wrote about last time, it is one of the fundamental components for good C++ programming. Francis describes it as the dimmest of 'smart pointers.' I am not sure that it even deserves that much, but in an exception handling environment it does provide a minimal tool for restricting loss of dynamic resources when an exception is thrown over them.

The problem

In your early experience of C++ you were taught that you could write such things as:

```
Mytype * mt;
unsigned int size;
cout << "How many Mytype objects?";
cin >> size;
mt = new Mytype [size];
```

The problem is that you are now responsible for the destruction of the array pointed to by *mt*. What happens if an exception is thrown during the lifetime of the dynamic array and is caught earlier than the scope of *mt*?

The answer is simply that the program has just leaked a chunk of memory. Even if you use debugging tools and detect this leakage, you need more than a pointer to cure the problem. Some operating systems will recover the lost memory when the program terminates, some will not (Windows 3.1 for example). Even the systems that recover the memory on program termination do not help with the possible continued leakage from a long running program.

Have you noticed that systems running Windows 3.x steadily slow down (an have increased disk activity) during a working day? That is symptomatic of memory leakage, the increased disk activity is because the shrinkage in available RAM is causing increased paging to virtual memory.

Somewhere in Bjarne Stroustrup's *The C++ Programming Language* he writes about encapsulating resource allocation so that a destructor will be called to clean up. A good example of this concept is the difference between file handling in C and C++. In C you should be careful to close files before the file-handle (*FILE* *****) ends its life. If you do not, you will have to rely on the clean-up provided by *exit***()** to close open files. As some systems only allow a limited number of files to be open at one time, this can cause unpleasant problems. The use of C++ *fstream*s removes this responsibility because the destructor of an *fstream* object closes the file.

What we need is a tool to extend this philosophy to general dynamic allocation of resources. The Standard C++ Library provides *auto_ptr* as a minimal tool to do this.

The definition of auto_ptr

```
template<typename X> class auto_ptr {
// private implementation details
public:
  explicit auto_ptr(X* p=0);
  template<typename Y>
      auto_ptr(auto_ptr<Y>&);
  template<typename Y>
      auto_ptr operator=
(<auto_ptr<Y>&);
  ~auto_ptr();
  X& operator*() const;
  X* operator->() const;
  X* get() const;
  X* release();
  void reset(X* p=0);
};
```

When **namespace** is available, this will be in namespace *std*.

I have used the new keyword **typename** for the template parameters because I think it is better

than the currently overloaded use of **class**. It was introduced into the language to allow disambiguation of names used in templates where the parser could not determine if an identifier was the name of a type or the name of an object. Having got it in the language, it was obviously better to be able to write **template<typename** *X*> instead of **template<class** *X*>. The standards committees made the obvious change (it couldn't break existing code, and it made new code more readable - I wish they would follow through and use it consistently in the C++ working paper).

*So do I! It's been a pet peeve of mine since we adopted **typename** and if I'd had the time, I would have introduced it in the examples in clause 14 [temp] when I was editing it earlier this year! – Ed.*

There is a second new keyword in the above code: **explicit**. This was introduced to obviate the need to use horrible hacks to stop constructors taking single arguments creating implicit type conversions. For example, without the **explicit** qualification of the *auto_ptr* constructor, the compiler could create an *auto_ptr* from any raw pointer with disastrous consequences. Consider the following code:

```
void fn (const auto_ptr<int> & handle){
// do something
}
void call_fn(){
       int * array=new int[1000];
       fn(array);
       // etc.
}
```

Because the parameter of *fn* is a **const &**, the compiler can try to create a temporary *auto_ptr***<int>** from *array*. Without the **explicit** qualification on the *auto_ptr* constructor, it will succeed. The result will be that the destructor of the temporary that *handle* is bound to will delete the storage pointed to by *array*. The sooner **explicit** is supported by our compilers the better.

Note: the result may be worse than this – see The Harpist's comments on auto_ptr and arrays below – Ed.

Note that the default value for an *auto_ptr* is to contain a null pointer.

Both the copy constructor and the copy assignment may appear to be a little strange. Your first reaction may be the same as mine and assume that someone has got their *X*'s and *Y*'s mixed up. This is not the case. The *Y* represents either an *X* or a class derived from *X* for which **delete(***Y****)** works. I am not sure how much burden that places on the programmer and how much compilers will be able to detect. To some extent this may depend on the quality of the implementation.

The member templates will fail to instantiate unless Y is an appropriate type, i.e., the compiler will detect this – Ed.

Before I continue with describing these copy facilities, I need to explain a little about the semantics of *auto_ptr* and the way these are supported.

There must only be one active *auto_ptr* for any object, anything else would result in the potential for multiple attempts to destroy an object.

Until changed by *reset***()** an *auto_ptr* will hold the address (pointer) that was specified by the constructor that created it, or by the most recent use of *reset***()**.

*get***()** returns this held value. In other words *get***()** always returns the most recent address stored in *auto_ptr* by either a constructor or a call to *reset***()**.

*release***()** is a special case, it returns the remembered value and then overwrites this with the null pointer. In other words, after calling *release***()** an *auto_ptr* will behave as if *reset***(0)** had been called.

*reset***()** replaces the remembered pointer with the new one passed as an argument (or to null if there is no argument in the call).

When an *auto_ptr* is cloned by copying, the original is changed by calling *release***()** on it. That is, the *auto_ptr* being copied now contains a null pointer while the copy now contains the original pointer.

A similar process is carried out for copy assignment, except that if both operands of the assignment hold the same pointer, no change takes place. In other words, if you have already messed it up by having two *auto_ptr*s responsible for a single object, an assignment will not help.

Think of *auto_ptr*s as being members of a relay team, only one can carry the baton at any one time and anyone crossing the finish line with the baton causes the baton to go out of use (i.e., deletes the contained pointer).

The other two member functions (**operator*()** and **operator->()**) are the standard functions required of any type of smart pointer. They ensure that *auto_ptr*s can be used as pointers where appropriate.

Using auto_ptr

As long as you keep a few simple concepts in mind, it is easy to use *auto_ptr*s. The main rule is that you should only use them in three cases:

1. to hold newly created dynamic resources,
2. to pass responsibility for dynamic resources forward to a function via a parameter (an unusual circumstance, but sensible if you do not intend to use the resource after return from that function) and
3. to pass ownership back on return from a function.

Strictly speaking you can pass references to an *auto_ptr* but there seems to be little practical reason for doing so. Remember that the primary reason for *auto_ptr* is to ensure that memory is returned when a dynamic object is finished with.

There is one interesting feature of using *auto_ptr*s: you will find it tempting to use them to handle dynamic elements of other objects. For example instead of writing:

```
class Name {
       char * name;
public:
       ~Name(){ delete [] name; }
       // rest of public interface
};
```

You might write:

```
class Name {
       auto_ptr<char> name;
public:
       ~Name(){}
       // rest of public interface
};
```

Tempting it may be, but it is wrong because *auto_ptr* holds the address of a single object, not an array. In other words its destructor executes **delete** *get***()** and *not* **delete[]** *get***()**. Even if your use does not fall into that trap, you still have gained little if anything. Using *auto_ptr* to access an object will be slightly slower because of the requirement to convert to a raw pointer when using it (i.e., by executing the appropriate operator function). Remembering to release dynamic resources in a destructor isn't that much of a burden.

Or use vector or string – Ed.

You would be much better to confine uses of *auto_ptr* to places where nothing else will release resources when an exception is thrown. I suppose it is possible that a complicated class with several dynamic elements might benefit in the case where the construction of an object fails, but when you get to that level of C++ programming you will be writing articles instead of reading them.

A typical use of *auto_ptr* might be:

```
void fn(size_t size) {
  if (size) {
    try {
      auto_ptr<vector<T> > vt(
                    new vector<T>(size)
);
      // code using vt
    }
    catch (bad_alloc) {
      // handle out of memory
    };   // see below for the warning
         // about putting this semicolon
         // here!
  }
}
```

Note that I have enclosed the actual allocation in a **try** block so that I can handle out of memory directly. Unless I have some specific exception that I want to handle if thrown by the subsequent code, I do not need to place it in a **try** block, nor do I need to explicitly release the dynamic *vector* because *vt* will be destroyed at exit from the **try** block (either normally, or because an exception is thrown through it) and that will destroy the dynamic *vector*.

Also note that the *auto_ptr* is for a *vector<T>* not just a *T*. There is one other potential catch, the closing angle brackets of templates must be separated by at least one character. If you forget this you will get an obscure error because the result will be parsed as a right shift operator.

Another point often missed by those writing their first catch clauses for a **try** block is that a semicolon closes the set of catches. Get out of the habit of adding semicolons after closing braces unless they are needed. Some of the time they will just be redundant but much of the time they will have significance.

Well I think that about covers it for this time round. If you have a compiler that supports *auto_ptr* and *vector* get into the habit of using them instead of raw pointers and C-style arrays. Don't pass *auto_ptr* by value unless you mean to pass responsibility for the dynamic resource to

the called function. The sooner you develop a coding style that uses these extra facilities the more robust your code will be.

The Harpist

An interesting aside on the >> vs > problem: Bjarne Stroustrup heard many complaints that it was "obvious" that >> in a nested template argument list should close both templates and so he tried to get the committee to accept a lexical change to allow this – a "hack". Various implementors said that users did indeed complain about it but typically each user only complained once, i.e., once it was explained to them that >> was an operator, they accepted it. The committee decided overwhelmingly that the problem was user education not a language "bug" – Ed.

Functionoids

by Jon Jagger

A functionoid is a class pattern which allows you to create objects that look and behave like functions. For example...

```
#include <iostream.h>
class Functionoid {
public:
  int operator()() { return 42; }
};

int main() {
  Functionoid func;
  for (int i = 0; i < 8; i++) {
    cout << func() << endl;
  }
  return 0;
}
```

As an example, consider using *rand***()** in C.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  printf("%d\n", rand());
  return 0;
}
```

The problem with this of course is that you haven't seeded *rand***()** by calling *srand***()**. To get round this you can quickly end up with code like this...

```
int random() {
  static int firstTime = 1;
  if (firstTime) {
    firstTime = 0;
    srand((unsigned int)time(0));
  }
  return rand();
}

int main() {
  printf("%d\n", random());
  return 0;
}
```

This technique of using a **static** to run first-time-only code is not multi-thread safe. C++ can do better...

```
#include <stdlib.h>
#include <time.h>

class TRandom {
public:
  int operator()() { return rand(); }
private:
  static unsigned int seed;
};

static int seeder() {
  time_t now = time((time_t*)0);
  struct tm * utc = gmtime(&now);
  unsigned int sec = utc->tm_sec;
  unsigned int min = utc->tm_min;
  unsigned int hour = utc->tm_hour;
  unsigned int seed =
            ((hour * 60) + min) * 60 +
sec;
  srand(seed);
  for (int i = 0; i < 256; i++) {
    (void)rand();
  }
  return 0;
}

int TRandom::seed = seeder();
```

Note that I do not use *srand***()** in the obvious manner...

```
    srand((unsigned long)time(0));
```

This is because *time***()** returns a *time_t*, which Standard C says is an arithmetic type. That means it could be a **float** or a **double**. Unlikely perhaps, but the very fact that the standard says it's legal strongly suggests that at the time the standard was ratified there was at least one compiler that didn't use an integer type. The cast could be a **float** / **double** to **unsigned int** which can be risky. A 100% portable version is more convoluted...

```
// first get a plain time_t
time_t now = time((time_t*)0);
// convert it to a struct tm.
struct tm * utc = gmtime(&now);
// convert a few int fields into
unsigned
// ints. these casts are safe because
the
// range of the fields is so small.
unsigned int sec = utc->tm_sec;
unsigned int min = utc->tm_min;
unsigned int hour = utc->tm_hour;
// create a seed. Arithmetic may
overflow,
// but overflow in unsigned integers is
// safe
unsigned int seed =
            ((hour * 60) + min) * 60 +
sec;
// and finally...
srand(seed);
```

You'll also note that I make a some calls to *rand***()** to get things underway. I have read that early values of *rand***()** are "not very random".

Lastly you can create a convenience function...

```
int random() {
  TRandom r;
  return r();
}
```

...since there is no reason for the user to know what is after all an implementation detail.

Jonathan Jagger
jonj@dmv.co.uk

I look forward to more articles on the subject of functionoids (although I prefer to call them functors, which is closer to the Latin "fungor" from which "function" is derived). This is a very useful technique with many applications – Ed.

Return from a member function

by Francis Glassborow

I seem to be in good company by having many of my member functions (particularly the *set* / *put* functions) return **void**. It is a short word, quick to type and follows a common style for global functions.

However, member functions are not global so perhaps there is a better choice than that popular with so many authors.

After giving the matter some thought, I think that the answer is yes, there is a better choice. Before I write up my answer I thought I would give the rest of you a chance to think about it. Put your ideas on paper and the best reasoned answer (even if entirely different from mine) will get a copy of *More Effective C++* by Scott Meyers.

Francis Glassborow
francis@robinton.demon.co.uk

Time please, ladies and gentlemen

by Jon Jagger

I have had the good fortune to use C++ for a small UNIX project at work for the last couple of months. Part of this project requires timed scheduling. I'd like to share what I found, partly because I've never seen much written about what follows, partly in response to Sean's and Francis' request for more "simpler" material, and partly to see if I get any feedback.

I started by looking at the basic types that *time.h* provides. They are of course *time_t* and **struct** *tm*. These are very different creatures. If you look at the C standard you'll see that *time_t* is an arithmetic type. That means it could be a **float** or **double**. Unlikely but legal. Also the standard says that the encoding of the *time_t* values is unspecified. That means that strictly speaking you cannot perform comparisons or arithmetic on *time_t* variables. Not looking so good thus far. Moving on, **struct** *tm* looks altogether more accommodating.

```
struct tm {
  int tm_sec;   // 0..62
  int tm_min;   // 0..59
  int tm_hour;  // 0..23
  int tm_mday;  // 1..31
  int tm_mon;   // 0..11
  int tm_year;  // 0..X == 1900..1900+X
  int tm_wday;  // 0..6, days since
sunday
  int tm_yday;  // 0..365, day of year
  int tm_isdst; // daylight saving time
};
```

These can occur in any order, and a **struct** *tm* can contain additional fields. Two things caught my attention. The first was "why is the *tm_sec* range from 0..62?" I found the answer: it's to allow for up to two leap seconds. No one realised that you can't have two leap seconds in the same year, let alone the same minute. The other was the *tm_isdst* field. Daylight Saving Time, DST is concerned with the clocks going forward or backward.

A leap second was added to the end of 1995 – Ed.

Clocks going forward

In the UK, this year, the clocks went forward one hour at exactly 1AM on March 31st. This means that if you were running this program...

```
#include <iostream.h>
#include <time.h>

int main() {
  for (;;) {
    time_t now = time(0);
    struct tm * loc = localtime(&now);
    cout << now << " " << asctime(loc));
  }
  return 0;
}
```

at that time, you would have seen something like this...

```
832632600 Sun Mar 31 12:59:59 1996
832632601 Sun Mar 31 02:00:00 1996
```

As an aside, note that in C you quickly get into a mess when trying to print out a *time_t* value. You don't know its type, so you have to pick a cast...

```
printf("time_t == %lu\n",
                (unsigned
long)time(0));
```

Clocks going backward

In the UK this year, the clocks will go backward one hour at exactly 2AM on October 27th. Once again, this means that if you are running the above program at that time you will see something like this...

```
853426100 Mon Oct 27 01:00:00 1996
853426101 Mon Oct 27 01:00:01 1996
......
853429698 Mon Oct 27 01:59:58 1996
853429699 Mon Oct 27 01:59:59 1996
853429700 Mon Oct 27 01:00:00 1996
```

time_t or not time_t

The point to note from these observations is that when DST occurs, the **struct** *tm* jumps, but the *time_t* value increments as normal. The conclusion is that you should schedule based on the *time_t* value and not the **struct** *tm* contents. Say you tried to schedule using the **struct** *tm* fields. The dangers are firstly that you might miss a scheduled time if it occurs during a lost hour when the clocks go forward, and secondly that you might repeat a schedule if it occurs during the duplicated hour when the clocks go backward. I chose to make the constructors take **struct** *tm* like fields however. This is because humans do not work well with *time_t*s, and also so *TimeStamp*s can be created in local time.

```
#include <time.h>
class TimeStamp {
public:
  TimeStamp();  // now
  TimeStamp( int hour,
             int minute,
             int second );  // today
  TimeStamp( int year,
             int month,
             int day,
             int hour,
             int minute,
             int second ); // specific
day
  //...

  bool operator == (
             const TimeStamp & rhs )
const;
  bool operator <  (
             const TimeStamp & rhs )
const;
  // ...

  bool neverOccurs() const;
  bool occursTwice() const;
  bool hasOccurred() const;

  void setDST( bool dst );
private:
  struct tm m_Local;
};
```

And a simplified (no schedule sorting) pattern of use goes like this...

```
vector<TimeStamp> schedule;
// get constructor parameters (eg from
GUI)
// and construct TimeStamp ts...
if (ts.neverOccurs()) {
    // refuse to accept. Tell user why.
    // suggest "nearest" alternative?
} else if (ts.occursTwice()) {
    // issue message to user..."stamp"
    // occurs during the hour affected
by
    // the clocks going backwards.
    // "stamp" will occur twice! Do you
    // want to schedule before the
clocks
    // go back (the DST time), after the
    // clocks go back (the normal time),
    // or cancel?
    if (response == before) {
      ts.setDST(true);
      schedule.push_back(ts);
    } else if (response == after) {
      ts.setDST(false);
      schedule.push_back(ts);
    }
}
//...
if (!schedule.empty()) {
  TimeStamp & next = schedule.front();
  if (next.hasOccurred()) {
    commit(next);
    schedule.pop_front();
  }
}
```

I chose to do the *TimeStamp* comparisons using non-member comparison operators for **struct** *tm*s, which convert the *tm* into a *time_t* using *mktime***()**.

```
static time_t remakeTime( struct tm & st
) {
  time_t l_secs = mktime(&st);
  if (l_secs == (time_t)-1) {
    throw SystemFailure(__FILE__
                                "
mktime()");
  }
  return l_secs;
}

static int tm_compare(
    const struct tm & lhs,
    const struct tm & rhs ) {
  // mktime() in remakeTime() can alter
the
  // tm fields, so copies are required
to
  // preserve the constness.
  struct tm copy_lhs = lhs;
  struct tm copy_rhs = rhs;
  // convert tm's to time_t's.
  time_t t_lhs = remakeTime(copy_lhs);
```

```
  time_t t_rhs = remakeTime(copy_rhs);
  // normally its a very bad idea to
  // compare a floating point value to a
  // manifest constant directly.
However,
  // difftime() guarantees to return a
  // whole number.
  double dif = difftime(t_lhs, t_rhs);
  if (diff < 0.0) return -1;
  if (diff > 0.0) return +1;
  return 0;
}

bool operator == (
    const struct tm & lhs,
    const struct tm & rhs ) {
  return (tm_compare(lhs,rhs) == 0);
}
bool operator < (
    const struct tm & lhs,
    const struct tm & rhs ) {
  return (tm_compare(lhs,rhs) < 0);
}
// etc etc
bool TimeStamp::operator == (
    const TimeStamp & rhs ) const {
  return (o_Local == rhs.o_Local);
}
bool TimeStamp::operator < (
    const TimeStamp & rhs ) const {
  return (o_Local < rhs.o_Local);
}
// etc etc
```

Note that *mktime***()** returns a UTC (Universal Co-ordinated Time, aka GMT). This is important, since you need to ensure that the following worst-case is well defined...

```
int October = 10;
TimeStamp before(1996,October,27,1,0,0);
// 1AM...
before.setDST(true);       // before
clocks
                           // go back
TimeStamp after(1996,October,27,1,0,0);
// 1AM...
after.setDST(false);       // after
clocks
                           // go back
assert(before < after);
```

The only non trivial part is testing for the DST clock changes. The following worked for me on a Solaris platform, but I wouldn't rate its chances on DOS/Windows very highly.

```
bool TimeStamp::neverOccurs() const {
  // copy is required to preserve const
  struct tm st = m_Local;
  // assume not in "lost" hour
  st.tm_isdst = -1;
  // remember the hour
  int hour = st.tm_hour;
  (void)remakeTime(st);
  // and if mktime() alters the hour
  // the assumption was false.
  return (st.tm_hour != hour);
}


bool TimeStamp::occursTwice() const {
  // copy is required to preserve const
  struct tm st = m_Local;
  int isdst0;
  int isdst1;
  // remakeTime once with DST off
  st.tm_isdst = 0;
  (void)remakeTime(st);
  isdst0 = st.tm_isdst;
  // remakeTime again with DST on
  st.tm_isdst = 1;
  (void)remakeTime(st);
  isdst1 = st.tm_isdst;
  // are both versions are valid?
  return (isdst0 == 0) && (isdst1 == 1);
}
```

Jonathan Jagger
jonj@dmv.co.uk

I certainly learnt a few things from Jon's article! I shall be less cavalier about using ***struct*** *tm in future – Ed.*

editor << letters;

FOR SALE

I think my (not so) old software may interest another member of the ACCU. I am selling the following for £80.00:

- Borland C++ 4.5 (CD and books)
- Borland PowerPack (32-bit extender for DOS) and books
- Borland Visual Solutions (v. 1.0) and books

I can deliver if not too far.

Remi Sellem
The Old Coach House
6 Monycrower Drive
Maidenhead
Berkshire SL6 1YQ
tel: 01628 222 51
101611.2501@compuserve.com

I tried Roger Wollett's code *[Overload 13]* with the Salford NT C/C++ compiler and linker. The same problem: missing

RList<**double**>**::***RLink***::#***RLink***()**

Regards

Bryan Colyer
bfc@vector.demon.co.uk

Sounds like the Salford compiler is also broken! Francis passed comment on this compiler in Overload 6.

My original message *[lost by the email ether – Ed]* was supposed to express disgust at Microsoft's "support" for STL in VC4.0 and ask if anyone knows of a good commercial implementation of STL for use with VC4 under NT. Things have moved on a bit since then but, for the record...

1. With language extensions disabled VC4 does not compile the STL code supplied with the compiler.
2. The **<bstring.h>** file supplied with VC4 requires a **<mutex.h>** for multi-threaded environments, but this is not supplied. (MFC 4.0, needless to say, has to be compiled for a multi-threaded environment.)

We have played with a copy of the *STL<ToolKit>* from ObjectSpace. It looks good to me and we will probably use it in-house. Any comments?

Phil Bass
pbass@rank-taylor-hobson.co.uk

I've heard good things about the ObjectSpace product but do any of the readers have hard experience with it?

Sean,

In the article *Some pitfalls of class design: a case study [Overload 13]* you (as editor) remark that

```
const Oid nigelsOid =
                "1.3.6.1.4.1.1503.22.1";
```

actually uses the dotted string constructor and the copy constructor (but that the copy constructor may be elided).

I always thought that the above was equivalent to

```
const Oid nigelsOid(
              "1.3.6.1.4.1.1503.22.1");
```

and I would be surprised if the copy constructor was used here.

Am I mistaken? I tried this out using MVC4 and the copy constructor wasn't used. I then made the copy constructor private since this should mean that if the copy ctor was required (even if not used) I would get an error message. I didn't. Is MVC4 broken here?

Cheers,

Colin Harkness

Unfortunately, Colin's mailer ate his reply address so I couldn't reply to this directly!

```
const Oid nigelsOid =

"1.3.6.1.4.1.1503.22.1";
```

is actually equivalent to:

```
const Oid nigelsOid(
    Oid("1.3.6.1.4.1.1503.22.1")
);
```

but most compilers optimise this to:

```
const Oid nigelsOid(

"1.3.6.1.4.1.1503.22.1");
```

However, the accessibility of the copy constructor should still be checked so, yes, MSVC4 is broken (as are many compilers in this particular case!).

Hi Sean,

Just reading the latest *Overload* and Roger Lever's article – you say at the end that the source will be available on our FTP site. Can you tell me what this is?

Also, I would like to express my appreciation of the work that goes into *Overload* by all the contributors. I have just sent my first contribution to *CVu*, hopefully one to *Overload* may follow in the not too distant future.

Thanks for your help in advance.

Best wishes,

Steve Watson
stevew@wallchart.com

Thankyou for your kind words Steve, I look forward to receiving your submission in due course!

The ftp site is at Demon:
`ftp://ftp.demon.co.uk/pub/accu`
and contains all the CVu source from issue 5.1 to issue 8.3 as well as Overload material which gets distributed with CVu's source material in the fullness of time.

Reviews

Words from Steve Oualline and Francis Glassborow after the unfavourable review of his book in a recent issue of *CVu*. Francis also reviews his favourite C++ development environment.

Practical C++ Programming
a response from Steve Oualline

Although this book was reviewed in CVu, both Francis and I feel the followup discussion belongs in Overload – Ed.

In his review Francis Glassborow states: "I do not like the author's approach and style..." I understand where Mr. Glassborow is coming from. He has strong opinions as to how things should be done and unfortunately *Practical C++ Programming* does not do things his way. His way concentrates on the theoretical and how things should be done. *Practical C++ Programming* is devoted to practical programming, that is getting things done.

A case in point is his criticism of the book for not using templates more extensively. The problem is that in the real world templates are simply not useful. I have used compilers from 6 different manufacturers and I have found 7 different implementations for templates. (Sun completely changed their implementation between version 3.0 and 4.0 of their compiler.)

Another example concerns the use of tax forms to demonstrate how virtual classes are used. The example was designed to show the reader how information could be laid out using a rather complex C++ construct. Tax forms provide the user with a concrete representation of the data.

Mr. Glassborow criticizes the code because among other things: "form_1040 is clearly neither a name nor a taxpayer." Frankly I fail to see why this class must be a name or a taxpayer.

It is derived from a class name that purports to represent a taxpayer. Hardly a sensible use of inheritance and certainly one that does not model "is-a" – Ed.

It is a class that contains information that must be filled in to complete a tax form. The other classes in this example also fall into this category. I realize that they do not fall into the categories set forth by Mr. Glassborow, but I also see no reason they should.

However, my rules for designing this class were: 1) Does it lay out the information clearly, and 2) Does it explain the concept of virtual classes clearly. I feel that it does a good job for both.

There is one problem with this example however. I tried to pick something that everyone would be familiar with, paying taxes. Unfortunately I threw in some form specific to the United States that do not translate directly to other countries.

Finally, Mr. Glassborow states "Borland have been supporting the new cast syntax and RTTI since version 4.0." For my book I did research Borland C++ Version 4.5 and found that it did support these features, but in a non-standard way. Non-standard don't count in my book.

I understand where Mr. Glassborow is coming from. He is concerned with pushing the design and architecture of the C++ language forward. I understand how he might be upset if I fail to use the latest features of the language or the newest design methodology.

I however, am a practical person. I tend to wait till a new language features if finished and settles does before using it. Also I tend to rely on more traditional programming methods and won't use new techniques till they have proven themselves.

There are places for people like Mr. Glassborow who are pushing the frontier of programming forward. My book is not designed for these people. My book is designed for the people who must make practical every day use of the C++ language to actually get work done.

I have edited the above to remove nearly a dozen spelling mistakes, but have left the original American spelling – Francis. You missed several which I deliberately left in – Ed.

Francis Glassborow responds

Steve Oualline wishes to designate my disapproval of his book as based on 'religious' differences, that is differences based on belief rather than fact. He declares he knows where I am coming from. I do not think he does. My primary concern is raising the quality of code. When I

look at a new book, one of the first things I examine is the quality of the author's code. If an author does not write good code based on reasonable analysis and design then it is unlikely that readers will manage to do so based on reading the book in question.

It is my contention that authors should write good code for examples that have been carefully chosen so that they not only show the use of the technique in question but are also based on good design. This does not have to be object-oriented design, though it should be based on some recognisable approach rather than a simple ad hoc "let's write some code." Sadly, I can find little indication that the author of this book (*Practical C++ Programming*) understands this.

Steve Oualline's coding style is typically mediocre. Of course there is a lot of code around that is worse than his, but few programmers will write better code as a result of emulating his examples. I note that he has the good grace not to try to defend the function that starts on page 471 of his book. I remind you that that function contains 15 **return** statements, a **switch** statement with 11 **case**s, 5 containing **while** statements, two of which have **break** statements. The function also contains a dozen **if** statements – two of which are inside a **while** statement nested in an **if** statement nested in a **case** clause of a **switch** statement. This is only an extreme example of a coding style that is the equivalent of badly written Basic spaghetti code.

On the issue of templates, he cannot even read what I write. The only sentence about templates in my review was '*While I can understand his reasons for saying very little about exceptions and templates I have reservations about his use of old C methods.*' Apparently he does not understand this. Whether he approves or not, pre-processor macros are extremely bad news in a C++ context where their invasive effects on class scopes etc. are very threatening. Simple template functions have been available for some years now and while extra power has been added, the simplest uses work as well today as they did a couple of years ago. While major companies such as Microsoft continue to sell 16-bit compilers that do not support templates it is reasonable for an author to explain how to achieve similar ends without templates, but where a programmer has a choice, the pre-processor option would rarely be appropriate. That is not just my opinion but the opinion of almost every professional C++ user and trainer.

At the Blackwell's meeting, the author admitted that **const** is not covered in his book *Practical C Programming* because he did not know it was part of ISO C. This is six years after the Standard was published. In such circumstances I take his opinions about Borland's implementation of RTTI with a large pinch of salt. Whatever the non-standard feature is that he is alluding to it does not justify his statement above and I think Borland would be deeply annoyed to see such a statement as they have always done their utmost to meet the requirements of the developing standard.

I do not accept Steve Oualline's rationale for his coding of the tax problem. The code was the result of completely inadequate design and is an example of the worst kind of hack-it-together coding. If he had looked more carefully he could have come up with a sensible example of using virtual base classes that would also have been a good example of an acceptable C++ coding style. I do not care what he does when writing code for himself or his employers, but I do care when he gives it as an example thereby encouraging others to emulate such poor design. This is the 1990's not the 1970's.

The tax example was just one of numerous places where the author has chosen completely spurious examples. Another is in his choice of an example for **friend** functions where he has a *stack* class declare a *stack_equal* global function. Absolutely wrong. There can be no conceivable reason for such a function to be anything other than a member function. Even if he was providing **operator==** it should still be a member function.

> *Well, many people prefer to make symmetric operators non-members – Ed.*

He completely misses a perfectly acceptable (even to me, though it is not my preferred method) use of **friend** to provide operator functions for complex numbers. Though in this case he curiously makes the data protected. Leaving aside the issue of whether data should ever be anything but private, complex is a value based concept and, as such a complex class should never be a base for a hierarchy.

One thing that became clear during the Blackwell's meeting is that Steve Oualline is abso-

lutely fanatical about comments (clearly I wish he were as fanatical about other aspects of good coding style). Unfortunately many of his comments are the useless kind that good programmers grow out of early in their programming lives. Let me give you a single example from page 220:

```
class int_set {
private:
  // ... whatever
public:
  int_set(void);   // Default
constructor
  int_set(const int_set & old_set);
                   // copy constructor
  void set(int value);
                   // set a value
  void clear(int value);
                   // clear an element
  int test(int value)const;
                  // see whether an
element
                  // is set
};
```

I do not know what you think, but I do not think these comments add anything. I'll accept the first two as possibly useful to those new to programming classes, but I think the next three are actually detrimental. *set***()** actually adds an element to the set, unless it is already there, *clear***()** removes an element if it is found (by the way, in both cases I would like to see a return value to indicate whether the function actually changed the set, but that is a style preference) and *test***()** checks to see if value is an element of the set.

Also note the C style of declaring a function with no parameters. If I was being pedantic I would argue that this is worse in the case of a constructor or destructor because these are nothing like C functions (at best they are true procedures, something that does not exist in C).

I could go on for pages, everywhere I look I see horrible code. On page 221 he discusses **const** members, and class-wide constants in particular. While he correctly gives the provision of these through an **enum** as an alternative he also suggests declaring a global constant as an alternative but never mentions a **static const** data member even though this is the topic he tackles in the next section.

The thing that concerns me is that this book has been published by O'Reilly & Associates, a publisher with a deservedly high reputation. The result will be that many thousands of programmers will have a very poor introduction to C++ which will result in much mediocre code. I am not concerned with cutting edge, bleeding edge or any other edge. I am concerned that programmers are helped to write solid, robust code. To do so they need an introduction to C++ that will give them a boost up the learning curve. I cannot believe that this book was technically reviewed by competent C++ programmers because I know that none of ACCU's C++ reviewers would have let it out.

If anyone else would like to review this book, I would be glad to let them have my copy. Please do not ask for it if you are not already a competent C++ programmer because it will do you more harm than good.

Finally, if you think that there is rather more acid in the above than usual, I deeply resent Steve Oualline's final paragraph.

The original review was published in *CVu* 8.1.

Francis Glassborow
francis@robinton.demon.co.uk

And The Editor says...

I have some additional comments to make on Steve Oualline's response:

He states that "in the real world templates are simply not useful" which shows he clearly has little understanding of real world programming. Templates are extensively used in major projects all around the world – compiler vendors will testify to this based on the number of user questions relating to templates!

I have looked at his book – a full review will appear in a future *Overload* – and my initial impression coincides with Francis': poor quality code illustrating barely thought out design.

Unfortunately for O'Reilly & Associates, their other recent C++ offering, *C++: The Core Language*, offers equally poor code although the intent of that book is noble – to teach a subset of the language to get C programmers up and running quickly. A full review of this book will appear in *Overload 15*.

Sean A. Corfield
overload@corf.demon.co.uk

My favourite C/C++ development package

by Francis Glassborow

I am not going to name this product for the moment. I hope you will understand why when I finally do so.

The XYZ product comes on a CD, requires a 32-bit operating system such as OS/2 or Windows NT and, as is increasingly the case, a large amount of RAM (16Mb+ depending on the operating system and the features you want to use). A full Windows NT installation takes nearly 400 Mbytes of disk space (now you know why my main machine runs twin 4Gb hard drives) and is only comfortable in 24Mb+ of RAM. However it will run off CD in which case the hard-drive requirement is small, though you do take a performance hit. When you have finished with the tutorials (more about those in a minute) you can trim quite a lot from the installation. Though it will run on a machine with anything from an 80386 upwards, it definitely needs a very fast 486 or a Pentium based machine for serious use. Though it is not listed as a requirement, I would not like to use this product on a 14" or 15" monitor. It makes heavy use of multiple windows, with large amounts of information, sometimes visual, in some windows. I think that a 17" monitor is the smallest for regular use of this product.

By now some of you will realise that this isn't a product for an amateur dabbler. You would be right, this is definitely a product for serious users.

I am only going to write about the Windows NT/Windows 95 version of the product here, but there are versions for a number of other platforms including Solaris and MVS. The importance of this is that the class libraries that underpin much of your development are largely portable (I wish I could say completely portable, but at least the residual problems of moving between such things as Windows NT and a Motif application are documented).

When you have installed the product on either Windows NT or Windows 95 (up to 20 minutes depending on the amount you install and the performance of your hardware) you should put aside time to work through the tutorials. These are the best I have ever experienced for a C/C++ product. There are tutorials on all aspects of programming with this product. Each leads you through several sections each divided into a number of steps. Each step is divided into two parts, what you should do and what the result should be. As long as you have a large enough monitor, you can have both the tutorial and your work on the examples on the screen at the same time. The tutorials are not perfect and you may have to look carefully at what you are getting to relate it to the specified results. It can be a little off-putting to get a view directly, when the tutorial assumes that you will get a different view and will need to press a button to select the one to be used.

The product supports several development styles.

You can work in a conventional style using a fairly standard editor. It is described as a parsing editor, but all that means is that it colour codes different features of your code. However the editor has another feature that can be more than a little useful, it supports some folding features. For example you can fold away all the bodies of functions. The editor comes with a range of preset styles so if you are used to using Brief, you can quickly switch to a familiar feel. You can also customise the editor in a wide variety of ways. There is nothing new in this, and a programmer might expect that such features were normal – unfortunately you know from experience that this is not actually the case.

At the other extreme, the product supports a full visual programming environment. I do not mean some dialogue based production of an application framework, I mean a genuine visual programming tool similar in concept to what users of products such as Visual Basic expect. I say similar in concept, because the actual mechanics are quite different. Do not expect to leap into visual programming after no more than a ten minute introduction. You will need to climb quite a long learning curve if you are to get best advantage from this development method, but the rewards in Rapid Application Development will be worth the invested time.

The Visual Builder is underpinned by an excellent open class library that supports a wide range of things that you may want to use including a very good set of tools for data access that are conveniently used from the Data Access Builder tool.

The product comes with a full range of all the other things that you expect from a development environment, debugger, profiler, browser etc.

By now those familiar with OS/2 will suspect that the product I am describing is IBM's VisualAge C++. They would be right, specifically VisualAge C++ 3.5 for Windows. Why, you ask, didn't I say so from the beginning? Well for all the actual publicity that IBM have generated for this important release, you might think that they did not want anyone to know about it.

Seriously, if you do substantial development in visual environments (MS Windows, X etc.) and/or database access programs or if you want to port applications easily between various 32-bit platforms you should check this product out.

At first sight you might think that IBM's release of this product for MS Windows platforms was a surrender to the might of MS. Nothing could be further from the truth. Indeed, if they had released this product a couple of years back (for NT) they might be in a stronger position today. Using VisualAge C++ largely insulates you from having to decide which platform you are developing for. It makes it easy to release OS/2, Solaris and AIX versions of the product that you develop for MS Windows. Of course it also works the other way but I suspect that will be less important.

VisualAge C++ is a good (though not perfect) product that I enjoy using. It is my first choice compiler. IBM should be shouting very loudly about its availability for MS Windows NT and Windows 95. While their open class library is not the same as the future C++ Standard Library, they will find it relatively simple to include that in a later release.

This is a product that deserves to succeed, but it needs much better marketing; technical quality is not enough.

Francis Glassborow
francis@robinton.demon.co.uk

News & Product Releases

This section contains information about new products and is mainly contributed by the vendors themselves. If you have an announcement that you feel would be of interest to the readership, please submit it to the Editor for inclusion here.

Hypersoft Europe

Adrian Lincoln of Hypersoft Europe provided the following announcements of new products from Rogue Wave and other library vendors.

Rogue wave's JFactory for Java

On February 26th Rogue Wave became the first company to release a visual application builder and code generator for Java – a key ingredient for those adopting this new language. This product JFactory enables developers to quickly create Java applications by dragging-and-dropping typical controls such as buttons, list boxes and menus.

JFactory provides a single design environment to manage all aspects of user-interface development. It includes a Project Manager, design windows, property sheets, and a palette of drag-and-drop controls. The developer arranges controls within the design window, sets the properties for each control, and associates controls with events or user-written code in a manner similar to a 4GL.

JFactory also provides the ability to test an application's interface, generate code, compile, and run the application. Since the environment also allows for incorporation of an editor, debugger, and compiler, all aspects of program creation can occur within JFactory.

Tools.h++ version 7

The latest version of Tools.h++, version 7, is now shipping. This new version has been enhanced to provide an object-oriented interface to the Standard C++ Library and extends it with new collection classes that are standard-compatible. Tools.h++ V7.0 is the latest release of this widely used class library for cross-platform development, and gives developers transparent access to the Standard C++ Library as a sub-set of its own extensive set of foundation classes.

A technical report called "The Standard C++ Library and Tools.h++" providing some more insight into the full power of this library combination is available.

ORBstreams.h++ for C++ CORBA developers

The good news for developers using IONA's Orbix is that they can now pass C++ objects easily using ORBstream.h++, which, based on the opaque mechanism in IONA's Object Request Broker allows objects to be passed objects by value in an IDLoperation. Previously, only types explicitly described in the CORBA IDL could be passed in an IDL operation.

ORBstreams.h++ makes it easy to pass opaque types by leveraging the powerful Tools.h++ virtual stream mechanism. All classes that are vir-

tual stream aware, including developers own code, may be passes as opaque types using ORB-streams.h++.

Objective Grid version 1.1 adds DAO and UNICODE/MBCS

This update to the Objective Grid MFC extension library provides support for DAO and UNICODE. The DAO support will provide developers with a richer option for Microsoft Access development than the ODBC class (note 32-bit version only). The UNICODE support with Objective Grid extends the capabilities of the grid making it easier to deploy MFC applications internationally.

SEC++ version 1.1 takes the library from 27 to over 40 classes

This powerful MFC extension class library SEC++ has been enhanced with the addition of many new classes. New additions include Dockable Document Interface, Floating Document Interface, Workspace classes, Popup calender classes, Colorwell classes, Encrypting CFile Derivatives, Bitmap Button, Intelligent Edit controls, Filesystem classes, and component gallery objects for each SEC++ component.

Many of these additions have been as a result of feedback from customers and hundreds of MFC developers over the past six months. Stingray Software plan to add 10-20 new classes per SEC++ release to give subscribers maximum value for their subscription.

Hypersoft Europe
adrian@hypersoft.co.uk

Take Five Software

In addition to a new release of the SNiFF+ integrated development environment, v2.2, Take Five provided the following items of note.

SNiFF+ for Java

Wake up and SNiFF+ the Java!

SNiFF+ provides an open interface for languages, enabling developers to work with C, C++ and Java simultaneously. This is an important differentiator from other Java development tools.

This new functionality, combined with the power of SNiFF+'s tools for code comprehension, team management and reverse engineering, will make SNiFF+ the ultimate programming environment for developing internet applications. In addition to the tools you are already familiar with, SNiFF+2 can be used with the Sun Java Developers Kit.

The new Java parser will be available free of charge to our current customers. Please call or email info@takefive.co.at for more information.

New platforms

SNiFF+2 is now available on Linux and Sinix. You can run SNiFF+2 on the following platforms:

- SunOS und Solaris (Sun SPARC)
- AIX (IBM RS/6000 und Power PC)
- HP/UX (HP RISC)
- Digital Unix (DEC Alpha)
- Irix (SGI)
- Novell UnixWare (PC)
- SCO Unix (PC)
- DEC Ultrix (DEC)
- Linux (PC)
- Sinix (SNI RM)

TakeFive Software is currently working on a version for Windows NT and Windows 95. Windows NT will be generally available by mid of June. The addition of new platforms enhances SNiFF+2's reputation as the most portable development environment.

If you are interested in being a part of our beta test program for Windows NT or Windows 95, please email sniff-beta@takefive.co.at for details.

Http://www.takefive.com

IDE announce Java and Unified Method support

Interactive Development Environments announces first OO Analysis & Design tool to generate and reverse engineer Java code; first to demo Unified Method v0.8

- Netscape integration enables immediate execution of generated Java applets
- Most advanced support of emerging Unified Method demonstrated at Software Development West

SOFTWARE DEVELOPMENT WEST, SAN FRANCISCO, March 26, 1996 – IDE, developer of the only object-oriented analysis and design (OOA&D) toolsets designed to support large development teams, today announced the industry's first support for both generation and reverse engineering of Java code, the new de facto standard language for the development of World Wide Web applications. Here, at Software Development West, IDE's StP/OMT-Booch OOA&D toolset is also the first to demonstrate complete support for the V0.8 of the Unified Method – the most advanced support provided to date by any vendor for the emerging standard method for OOA&D.

Large-scale Web Development

By keeping OO design models synchronised with Java code implementations through reverse engineering, developers can continually analyse the impact of code modifications. Web developers can also reverse engineer existing Java applications and reuse components in future applications. Because StP/OMT-Booch models are stored in a common, multi-user repository, reusable Java applets are available to developers throughout an organisation.

IDE will also support an integration with Netscape Navigator and the Applet Viewer in the Java Development Environment from Sun Microsystems, Inc. The Netscape integration takes Java code generated from StP/OMT-Booch models, compiles it, and sends a message to load the HTML page containing the Java applet into Netscape Navigator. This make iterative development of Java applets faster and easier.

Unified Method Support

StP's full support for the Unified Method V0.8 required merely incremental development, since the tool already provides more complete support for both the OMT and Booch methods, respectively, than any other toolset, and also integrates Jacobson Use Cases.

StP also enhances the Unified Method with a fully integrated Requirements Table Editor for collecting and tracking business rules through the OO lifecycle. The Requirements Table Editor lets developers collect business rules during the process of analysis, refine those rules through the lifecycle, and allocate and track them to ensure that they are satisfied completely by the resulting system. Allocation links enable developers to instantly navigate between the listing of business rules and the actual objeet that satisfy those business rules.

About StP/OMT-Booch

StP/OMT-Booch is a fully-featured, multi-user analysis and design environment with a shared central repository that supports entire teams through the full life cycle of application development. Using this repository as an integration link, StP/OMT-Booch assures the consistency, completeness and semantic correctness of all models – even across large project teams. Consistency checking also incorporates models from StP for Information Modelling (StP/IM), IDE's tool for database analysis and design and SQL code generation. The repository also allows users to browse model information easily and supports reuse in other design projects.

StP/OMT-Booch provides the industry's most complete automatic source code generation for C++, Ada83, Ada95, IDL, Forte, Smalltalk and reverse engineering for C++. Integration with VisualWorks® from ParcPlace-Digitalk, Inc. provides users with full forward and reverse engineering of Smalltalk code. In addition, StP/OMT is integrated with StP/T, a specification-based test case tool that automatically generates test cases from object and functional models of OMT.

Availability and Pricing

StP/OMT and StP/Booch 3.1 with Ada95 and Forte code generation is available immediately on Sun SPARC platforms running Solaris 2.4 and 2.5, and HP 9000's running HP-UX 9.05 or 10.01. StP/OMT and StP/Booch 3.1 are each priced at £8,950.00 per licence. The combined StP/OMT-Booch is available at £11,250.00. StP/OMT and StP/Booch 3.2 with Java code generation, reverse engineering and IDL reverse engineering will be available in June on these same platforms with the same pricing structure as version 3.1.

Interactive Development Environments Inc
+44 (0) 1483 579 000
http://www.ide.co.uk

Credits

Founding Editor

Mike Toms
miketoms@calladin.demon.co.uk

Managing Editor

Sean A. Corfield
13 Derwent Close, Cove
Farnborough, Hants, GU14 0JT
overload@corf.demon.co.uk

Production Editor

Alan Lenton
alenton@aol.com

Advertising

John Washington
Cartchers Farm, Carthouse Lane
Woking, Surrey, GU21 4XS
accuads@wash.demon.co.uk

Subscriptions

Barry Dorrans
2, Gladstone Avenue
Chester, Cheshire, CH1 4JU
barryd@phonelink.com

Distribution

Mark Radford
mark@twonine.demon.co.uk

Copyrights and Trademarks

Copy deadline

All articles intended for inclusion in *Overload 15* (August) should be submitted to the editor by July 22nd.