ISSN 1354-3172

Overload

Journal of the ACCU C++ Special Interest Group

Issue 11

December 1995

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

Subscriptions:
Membership Secretary
c/o 11 Foxhill Road
Reading
Berks
RG1 5QS
pippa@octopull.demon.co.uk

£3.50

Contents

Editorial **3**

Tools of the devil? 3

Thanks! 3

Software Development in C++ **5**

Classes and Associations 5

Java? Where is that? 15

The Draft International C++ Standard **18**

The Casting Vote 18

Some thoughts on linkage 19

Literally yours 21

Anonymously yours 22

C++ Techniques **23**

Simple classes for debugging in C++ – Part 2 23

A deeper look at copy assignment 28

Change of address 34

/tmp/late/ Generating constants with templates* 37

editor << letters; **39**

++puzzle; **40**

Date with a Design 40

Books and Journals **41**

Scientific and Engineering C++ 42

News & Product Releases **43**

SNiFF+2.1 43

Editorial

I suppose I should wish you all Merry Christmas and a Happy New Year.

This is the sixth issue of *Overload* this year – it's been hard work and a lot of fun. I hope you've been pleased with the year's material? If not, please contribute something you *would* like to read.

Tools of the devil?

Craftsmen rely on the tools of their trade. If a tool is faulty, they can take it back, get a replacement and carry on with their work. An inconvenience, but not likely to be a disaster. Their business won't necessarily go broke from poor tools. Perhaps more to the point, their tools market is mature enough and competitive enough that faulty tools are quite rare.

You know what I'm going to say next, of course. I can hear the words forming in your minds: software development tools regularly have faults. We learn to live with them, don't we? Compiler bugs can be infuriating, they cost us money, but we can usually work around them and get on with the job. Debuggers, testing tools, CASE software – we workaround, grumble and continue. And our business suffers a little each time, but we don't measure the loss and we let it pass.

But not all software tools are like this. Consider a substantial third-party library, perhaps a database class library or a GUI class library. Often, a decision to adopt one of these tools has to be made early on in the lifecycle of a project and the project soon comes to rely on the tool. What happens when faults are uncovered in such a library? Can you take it back to the vendor and get a replacement that works? Very unlikely ("That'll be fixed in the next release, next year."). Can you change vendors? Also very unlikely – interfaces are not mature enough to swap out one library and swap in another without fairly major changes to the client code.

Should you try to get your money back? How much have you lost: the cost of the tool, a percentage of the cost of development invested in your existing code... Lost time, however, may be more critical. Redevelopment using a new library might cause you to miss a window of opportunity in the market – what cost your lost business, or even the entire business itself?

I have just witnessed a project collapse due to the failings of a well-known, cross-platform GUI library. Since I heard the news, I have read several reports on the 'net that confirm these failings on other platforms that the library supposedly supports. A very expensive experience and I hope that the victim of this fiasco can recover and doesn't suffer too much from the lost market opportunity.

Is there anything our industry can do about this sort of thing? Tool vendors can take more care – so few companies actually exercise "best practice" that the age-old excuse that "bugs are inevitable in a product this complex" just won't wash: they *could* do better! As users, and victims, of these faulty products, we can be more publicly vocal about bad products. Write to the press (including *Overload*!), post information on the 'net, exercise your legal right to products that are "fit for purpose". Ask the vendor for a refund and if that fails, sue them. Delivering poor quality software tools must stop being financially viable for companies.

Thanks!

Welcome to the Kevlin Henney special edition of *Overload*! Thanks for contributing so much to this issue after I said that *Overload 11* was looking a little thin.

If the rest of you want such gratitude, start writing articles for the February issue!

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

FULL PAGE ADVERT GOES HERE!

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++.

David Davies follows up his article on OOA by looking at that part of the software process where design actually becomes code and The Harpist takes a brief look at Java.

Classes and Associations

by David Davies

Introduction

A previous article gave an overview of the OOA process and showed how a requirement specification was flowed down to high level design constructs[1]. The analysis exercise identified the classes, relationships between classes, class attributes and operations required to realise the specification. This article looks at some of the implementation issues of one of the outputs of the OOA activity – the Information Model and in particular, associations between classes. To recap, the Information Model, in Shlaer-Mellor terminology, identifies objects, their attributes and the static relationships between them. It is a way of capturing the semantics of the problem domain and organising the information into a formal structure.

Associations

In any sizeable application, separate independent classes collaborate to carry out a specific task. Classes that collaborate are said to be “associated” or “related”. Relationships may be binary (between two classes), ternary (between three classes) or of higher order. Higher order relationships are much more complex and are generally to be avoided. This article considers only binary relationships or associations.

Associations provide the means to link objects in a meaningful way. Each pair of instantiated objects can be considered as a unit or tuple reflecting a particular instance. An association between classes *Owner* and *Dog* would have instances: *Jack* and his dog, *Rover*; *Ray* and his dog, *Bonzo*.

Using an associative object to express an association is another way of modelling the same underlying concept. Since objects have properties, the associative object can be used to hold properties of the relationship. The association between *Owner* and *Dog* could be expressed through associative object *Licence* which in addition to linking instances of *Owner* and *Dog* would contain licence details such as serial number, date of expiry.

Generally an associative object is used if the tuple has subsidiary information, otherwise an association is used.

Cardinality

The cardinality of an association is the number of instances of a class that participate in an association and can be one-to-one, one-to-many or many-to-many. The example of the one-to-one association used in the article is the MP and constituency association. There can only be one MP representing a constituency and a MP may represent only one constituency. The one-to-many association is taken from a library scenario where a borrower can borrow many books but a book can only be borrowed by a single borrower at a time. The many-to-many association is demonstrated by the actor-play association. An actor can appear in many plays and a play has many actors.

Implementation considerations

There are several ways of implementing associations depending on the cardinality of the relationship and whether traverse is required from one end only or from both ends. If only one end needs to refer to the other an unidirectional traverse can be implemented, if both ends are involved then a bidirectional traverse is required. A unidirectional traverse is simpler to accomplish as only one end of the association has to hold information about the other and may be suitable when the reverse relationship is never, or only infrequently, required. Although it is possible to traverse in the opposite direction, it involves interrogating every object which has traverse information and determining if it is part of the association in question. A potentially lengthy and expensive operation.

Table 1 shows the various options discussed in this article. For the one-to-many association, three traverse conditions are considered; unidirectional from the “one” end, unidirectional from

Cardinality	Traverse	Technique	Listing
1-m	unidirectional from "1"	basic	Listing 1
1-m	unidirectional from "1"	using Container Class	Listing 2
1-m	unidirectional from "many"	with pointers	Listing 3
1-1	bidirectional	with pointers and friends	Listing 4
1-1	unidirectional	using Association & Dictionary Classes	Listing 5
1-1	bi-directional	using associate class	Listing 6
1-m	bi-directional	with pointers and friends	Listing 7
m-m	bi-directional	with pointers and friends	Listing 8

Table 1 Some methods of implementing associations

the “many” end and bidirectional. For the one-to-one association both unidirectional and bidirectional traverses are considered.

The high level constructs produced in the analysis stage require fleshing out to ensure that useable, robust and maintainable classes are produced. An analysis of a library system has identified that there is a one to many relationship between *Borrower* and *Books*. Using the OMT notation for describing classes (where a class is represented by a rectangle divided into three horizontal sections: the top section contains the class name, the middle section the attributes of the class and the lower section the member functions) the analysiś view of the classes and their relationship is shown in Fig 1A. Fig 1B shows the class designer’s view of the relationship, having to add constructors, destructors and access members in order to produce an implementable design, plus class libraries that provide standard reusable components for incorporation into the application.

A common problem in real life applications is the handling of groups of items. Developing data structures to hold and manipulate these items can take up a lot of development time. Borland C++ provides programmers with a robust collection of reusable container classes so that effort can be concentrated on the application and less on the implementation details.

The OO philosophy can be expressed as:

- Don’t build, buy
- Don’t invent, reuse.

With this in mind several of the implementations use the Borland container class library to illustrate how incorporating these can simplify application development.

And soon we will all have access to the powerful set of container classes from STL that will be available with every C++ compiler as part of the standard library – Ed.

Listings

A brief description of the salient implementation issues of each approach is given below. The programs were compiled using Borland C++ 4.0.

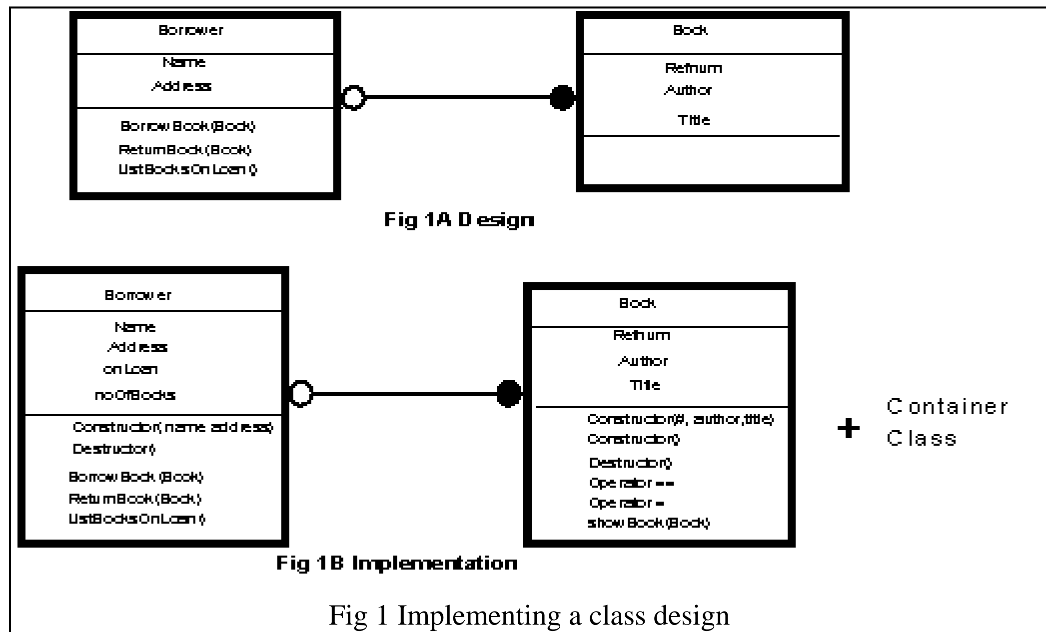

Fig 1 Implementing a class design

One-to-many association. Listing 1

This design approach is the simplest of the eight covered in this article. The *Borrower* class contains an array of *Book* objects which holds details of books on loan. It supports traverse from *Borrower* to *Books* and so it is easy to ascertain which books have been borrowed by a particular borrower. However, it is not so straightforward to determine which borrower has a particular book as it involves searching through all borrowers to find it. It is not a particularly efficient design as it holds objects (rather than pointers to objects) and an array is not a particularly good data structure when there are variable storage requirements. The size of the array is set to the maximum regardless of how many books the borrower has out on loan. It is also “hand crafted” and does not include any features utilising re-use of existing components.

One-to-many using Class Library. Listing 2

In a re-use frame of mind, the built-in *Container* classes can be incorporated to reduce the amount of hand-crafted code. Listing 2 uses a container class to provide data storage and so is an improvement on the first offering. The iterators provided with the container class make manipulation of the data easier. Again the container holds book objects, a better design approach would be to use an indirect container which holds pointers to book objects.

One-to-many using pointers. Listing 3

This provides unidirectional traverse from the many to the one. Each *Book* object holds a pointer to its associated *Borrower*. It enables the question “Who has book X?” to be easily answered as each *Book* object holds a reference to its *Borrower*. As a borrower may have many books on loan it follows that the same reference will have to be used by several *Book*s. Since only a single copy of a specific *Borrower* object can exist, pointers are necessary so that multiple instances of *Book*s can refer to the same *Borrower* object. Objects of book and borrower are instantiated and initially books have their borrower pointer set to zero. The *borrowBook* member sets the pointer to the appropriate borrower object. Returning a book is modelled by calling the *returnBook* member which resets the pointer to zero.

One-to-one with pointers and friends. Listing 4

Listing 4 shows an implementation for a bidirectional one-to-one association. Traverse is supported in both directions making it easy to efficiently move in both directions and to be able to answer such questions as “Who is the MP for the New Forest constituency?” or “Which constituency does Sir Patrick McNair-Wilson represent?”

A difficulty in implementing bidirectional traverse is that cross references have to be maintained between the two objects. One way to achieve this is by using friends. I know that the use of friends is frowned upon in some quarters, but taking a pragmatic view, I believe that there are some instances where friendship can facilitate understandable and maintainable design. Friendship is a controlled way of defeating encapsulation and granting another class access to the private data of another. The operative word is “granting” – a class can only be given friendship, it cannot claim it.

In this example each class has friend members to link or unlink associations between objects. The link and unlink functions update the attributes of the associated class in addition to their own class via private members to maintain data integrity. Instantiation and deletion of objects is performed separately from linking. Linking only sets the links between existing objects.

An alternative implementation, which maintains encapsulation, uses public members to update the book and borrower details. To maintain data integrity the members have to be used properly but there is no mechanism to enforce this. The class designer, therefore, has no control on how they are used. This type of implementation is not covered in this article.

One to one association using class libraries. Listing 5

As in listing 2 the design can be implemented utilising the class libraries supplied with the compiler. Listing 5 shows a one-to-one association implemented using the provided Association and Dictionary classes. The two classes *Tenant* and *Apartment* are associated via the *TIIAssociation* Association. The ‘I’s in the name indicate indirect storage is used, hence pointers are stored by the association object. The associations are then stored in *TIDictionary lettings*. After three entries associating a tenant with an apartment,

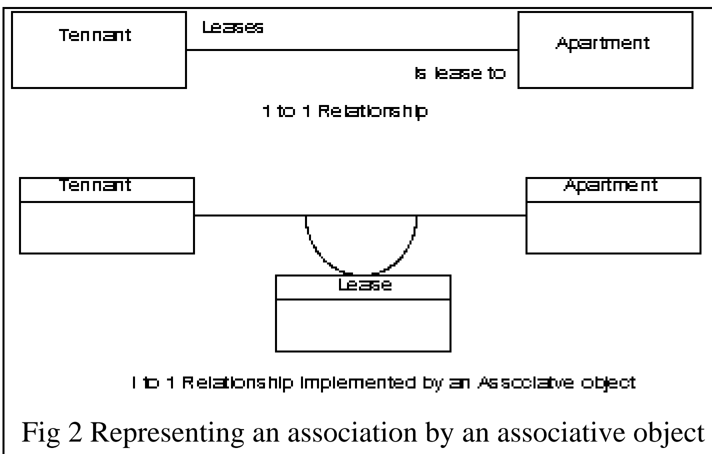

Fig 2 Representing an association by an associative object

the contents of the dictionary are displayed using the dictionary iterator function.

The Association class defines a relationship between two entities, the key and the value. In this example *tenant* is the key whilst *apartment* is the value. However the order could easily be reversed with *apartment* as the key. It is always the first parameter in the association declaration that is taken as the key. The class Dictionary is internally implemented with a hash table and so *Tenant* has to be derived from *String* to give access to *String*'s *HashValue* member function and so generate a hash value for the *Tenant* object.

Objects of type *Tenant* and *Apartment* are first created. Then they are linked into an association and added to the dictionary. A meaningful == operator has to be provided in the class that is used as the key in the association. This is required to ensure correct operation of the hash function.

One-to-one association with associative class. Listing 6

An association can be represented by using an associative object that contains references to identifiers in each of the participating instances. This is applicable when information about the association has to be held. The association between tenant and apartment is used as an example. There is a one-to-one association between *tenant* and *apartment* of *lease*, a tenant leases an apartment and an apartment is leased to a tenant. *Lease* can be implemented as an associative object when information on the lease such as duration, rent etc. has to be held. See fig 2.

In this example, associative object *lease* contains pointers to each object participating in the relationship. *tenant* and *apartment*. Reverse pointers to *lease* are held by *tenant* and *apartment*. This supports bidirectional traverse from *tenant* and *apartment*.

One to many bi-directional traverse. Listing 7

This is an implementation of a bidirectional one to many relationship for the library scenario. In this case it is possible to efficiently traverse in both directions. It enables the identification of the borrower of a particular book without having to interrogate every borrower. Each *Borrower* object holds an array of pointers to borrowed *Book* objects and each *Book* has a pointer to its current *Borrower*. Friend members make or remove links between *Borrower* and *Books*. Objects are created and destroyed at the start and end of the main program. The general design is shown in Fig 3.

The slots in the array holding pointers to book objects are initially set to zero. The *BorrowBook* member looks for the first free slot in the array and inserts the new pointer in that location.

Many-to-many bi-directional traverse. Listing 8

The thespian scenario is used as an example of a bidirectional many to many relationship. An actor can appear in many plays and a play has a cast of many actors. Unlike the example library where there was a requirement limiting the number of books taken out by a borrower at any time here the cardinality of the relationship is undefined, the number of entries depends on the popularity of the actor. Instead of using a fixed size data structure such as an array, an extensible data structure which can grow as the number of entries increases is required. A linked list is a suitable data structure. These come in many flavours, single, double, sorted etc but the example uses a simple single linked list. The design uses a variant on the one-to-many implementation shown in listing 7. In this case each object on both sides of the association holds a collection of pointers to linked objects on the other side. The links are balanced, both ends are updated contemporaneously.

The *Clist*, holding cast details, and *RList*, holding actor details, are based on a single linked list of **void** pointers, *Blist*. Such a list is considered to be type unsafe insofar as it can hold pointers of any type. Type safety is achieved by ensuring that the insert and remove members of *Clist* and *Rlist* will only accept pointers of the appropriate type.

Friend members update the links on both sides of the association and data integrity is ensured by using private members to modify the link details.

Conclusion

This article has explored some of the ways that relationships between classes can be implemented. Variations on the techniques described can be used in real life applications. Currently there are a multitude of books on C++ programming covering the 'nuts and bolts' of writing code and a vast number which have either OOA or OOD in their title but very few which cover the transition from analysis to code. An article in a recent issue of EXE [2] touches on this issue. One of the few books to explore the relationship between object modelling and C++ programming is "Inside the Object Model" by David Papurt [3].

David Davies

References

[1] Overload 8, "OOA- The Shlaer-Mellor Approach", David Davies

[2] EXE vol. 10 Issue 1 June 1995, pp 47-54, "Seamless, but open to interpretation " Mary Hope

[3] SIGS Books, "Inside the Object Model", David Papurt 1995 ISBN 1-884842-05-4

I have decided to break with tradition by including the entire source code verbatim. This has been done because quite a large part of David's article refers directly to the different approaches used in the source code.

The code will be on a future CVu disk and later on Demon's ftp site. Once you have the code in your hands, I would welcome contributions which recast David's code using STL and comment on benefits and disadvantages (e.g., the STL array container – vector – can change size; the list container does not offer a choice of singly-linked implementation; the dictionary container – map – is not hash-value based) – Ed.

```
//Listing 1
#include <iostream.h>
#include <string.h>

class Book;

class Book
{
public:
        Book(int refnum, char* author,  char*
title);
        Book();
        Book& operator=(const Book& b);
        void showBook(Book book);
        int GetBookNo(){return o_refnum;}
private:
        int o_refnum;
        char o_author[30];
        char o_title[30];
};

class Borrower
{
public:
        Borrower(char* name, char* address);
        void borrowBook(Book book);
```

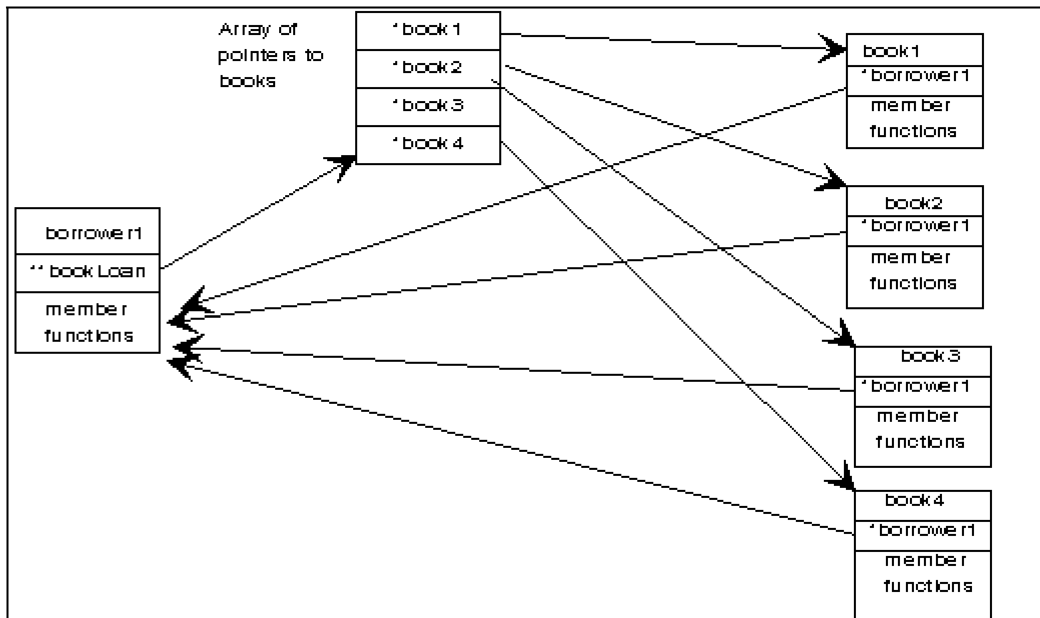

Fig 3. 1-M Bidirectional traverse

```
        void returnBook(Book book);
        void listBooksOnLoan();
private:
        char o_name[20];
        char o_address[30];
        Book onLoan[5];
        int noOfBooks;
};

Book::Book(int refnum, char* author,  char* title)
        {
        o_refnum = refnum;
        strcpy(o_author, author);
        strcpy(o_title, title);
        }

Book::Book()
        {
        o_refnum = 0;
        strcpy(o_author,"");
        strcpy(o_title, "");
        }

Book& Book::operator=(const Book& b)
{
        o_refnum = b.o_refnum;
        strcpy(o_author, b.o_author);
        strcpy(o_title, b.o_title);
        return *this;
}

void Book::showBook(Book book)
{
cout << " Book number: " << book.o_refnum <<
    ", Author: " << book.o_author
    << ", Title: " << book.o_title << "\n";
}

Borrower::Borrower(char* name, char* address)
        {
        strcpy(o_name, name);
        strcpy(o_address, address);
        noOfBooks = 0;
        int i = 0;
        Book t;
        while (i < 5)
                onLoan[i++] = t;
        }

void Borrower::borrowBook(Book book)
        {
        if(noOfBooks == 5)
                cout << " You have borrowed the "
                        " maximum permited number
"
                        "of books\n";
        else
                {
                int i = 0;
                while (onLoan[i].GetBookNo() !=0)
                        i++;
                onLoan[i] = book;
                ++noOfBooks;
                }
        }

void Borrower::returnBook(Book book)
        {
        if(noOfBooks == 0)
                cout << " No books on loan\n";
        else
                {
                int i =5;
                Book t;
                while(i-- > 0)
                        if(onLoan[i].GetBookNo()
                          == book.GetBookNo())
                                {
                                cout << " Book "
                         <<  book.GetBookNo()
                         << " returned\n";
                                onLoan[i] = t;
                                }
                noOfBooks--;
                }
        }

void Borrower::listBooksOnLoan()
        {
        Book bk;
        for(int l =0; l<5; l++)
                if(onLoan[l].GetBookNo() != 0)
                        bk.showBook(onLoan[l]);
        }

int main()
{
Borrower fred("F James", "40 Riverside");
```

```
Borrower john("John Smith", "345 Station Rd");
Book book1(1234, "Oscar Wilde",
         "Picture of Dorien Gray");
Book book2(12345, "Charlotte Bronte", "Jane Eyre");
Book book3(2345, "Oscar Wilde",
         "Importance of Being Earnest");
Book book4(2348, "C Dickens", "A Christmas Carol");
Book book5(5348, "C Dickens", "A Tale Of Two
Cities");
fred.borrowBook(book1);
john.borrowBook(book3);
fred.borrowBook(book2);
john.borrowBook(book4);
fred.borrowBook(book5);
cout << "List Fred's books \n";
fred.listBooksOnLoan();
cout << "List John's books \n";
john.listBooksOnLoan();
fred.returnBook(book2);
cout << "List Fred's books \n";
fred.listBooksOnLoan();
return 0;
}
```

```
//Listing 2

#include <iostream.h>
#include <string.h>
#include <classlib\arrays.h>

class Book
{
public:
        Book(int refnum, char* author,  char*
title);
        Book();
        Book& operator=(const Book& b);
        int operator == (const Book& b)
        {return o_refnum == b.o_refnum;};
        void showBook(Book book);
private:
        int o_refnum;
        char o_author[30];
        char o_title[30];
};

typedef TArrayAsVector<Book> bookArray;
typedef TArrayAsVectorIterator<Book> bookIterator;

class Borrower
{
public:
        Borrower(char* name, char* address);
        void borrowBook(Book book);
        void returnBook(const Book& book);
        void listBooksOnLoan();
private:
        char o_name[20];
        char o_address[30];
        bookArray *onLoan;
        int noOfBooks;
};


Book::Book(int refnum, char* author,  char* title)
        {
        o_refnum = refnum;
        strcpy(o_author, author);
        strcpy(o_title, title);
        }

Book::Book()
        {
        o_refnum = 0;
        strcpy(o_author,"");
        strcpy(o_title, "");
        }

Book& Book::operator=(const Book& b)
{
        o_refnum = b.o_refnum;
        strcpy(o_author, b.o_author);
        strcpy(o_title, b.o_title);
        return *this;
}

void Book::showBook(Book book)
{
        cout << " Book number: " << book.o_refnum
        << ", Author: " << book.o_author
        << ", Title: " << book.o_title << "\n";
 }

Borrower::Borrower(char* name, char* address)
        {
        strcpy(o_name, name);
        strcpy(o_address, address);
```

```
        noOfBooks = 0;
        onLoan = new bookArray(10);
        }

void Borrower::borrowBook(Book book)
        {
        if(noOfBooks == 6)
                cout << " You have borrowed the "
                "maximum permited number of "
                "books\n";
        else
                {
                onLoan->Add(book);
                noOfBooks++;
                }
        }

void Borrower::returnBook( const Book& book)
        {
        if(noOfBooks == 0)
                cout << " No books on loan\n";
        else
                        {
                        onLoan->Detach(book);
                        noOfBooks--;
                        }
        }

void Borrower::listBooksOnLoan()
        {
        bookIterator i(*onLoan);
                Book bk;
                while(i)
                        {
                        bk = i++;
                        bk.showBook(bk);
                        }
                }

//uses same main() as listing 1
int main()
{
Borrower fred("F James", "40 Riverside");
Borrower john("John Smith", "345 Station Rd");
Book book1(1234, "Oscar Wilde",
         "Picture of Dorien Gray");
Book book2(12345, "Charlotte Bronte", "Jane Eyre");
Book book3(2345, "Oscar Wilde",
         "Importance of Being Earnest");
Book book4(2348, "C Dickens", "A Christmas Carol");
Book book5(5348, "C Dickens", "A Tale Of Two
Cities");
fred.borrowBook(book1);
john.borrowBook(book3);
fred.borrowBook(book2);
john.borrowBook(book4);
fred.borrowBook(book5);
cout << "List Fred's books \n";
fred.listBooksOnLoan();
cout << "List John's books \n";
john.listBooksOnLoan();
fred.returnBook(book2);
cout << "List Fred's books \n";
fred.listBooksOnLoan();
return 0;
}
```

```
//Listing 3
#include <iostream.h>
#include <string.h>

class Book;

class Borrower
        {
        public:
                Borrower(char* A);
                ~Borrower() { delete aName;}
                void showBorrower()
                {cout << aName << "\n";}
        private:
                char* aName;

        };

class Book
        {
        public:
                Book(char* T);
                ~Book()
                {LoanedTo(0); delete aTitle;}
                void  LoanedTo( Borrower* br )
                {a= br;}
                Borrower* onLoanTo() {return a;}
                void BorrowBook(Borrower &a,
                                Book &b);
                void ReturnBook(Borrower &a);
        private:
                Borrower *a;
                char* aTitle;
        };

Borrower::Borrower(char* A)
        {
        aName = new char[strlen(A)+1];
        strcpy(aName, A);
        cout <<  "Borrower " << aName
                 << "\n";
        }

Book::Book(char* T)
        {
        a = 0;
        aTitle = new char[strlen(T)+1];
        strcpy(aTitle, T);
        cout <<  "Book " << aTitle << "\n";
        }
void BorrowBook(Borrower & A, Book & B)
        {
        if(!B.onLoanTo())
                {
                        B.LoanedTo ( &A);
                }
        }

void ReturnBook (Book & A)
        {
        if(A.onLoanTo())
                {
                A.LoanedTo(0);
                }
        }

int  main()
{
Book oliverTwist("Oliver Twist");
Book eyre("Jane Eyre");
Book pictureOfDorienGray("Picture of Dorien Gray");
Borrower smith("John Smith");
Borrower jones("David Jones");
BorrowBook(smith,oliverTwist);
BorrowBook(smith,pictureOfDorienGray);
BorrowBook(jones,eyre);
cout << " Traverse from book to Borrower\n";
cout << "Who has  Oliver Twist?  ";
Borrower *p=oliverTwist.onLoanTo();
p->showBorrower();
cout << "Returns book\n";
ReturnBook(oliverTwist);
BorrowBook(jones,oliverTwist);
cout  << "Book now borrowed by Jones \n";
cout << "Who has  Oliver Twist?  ";
Borrower *q=oliverTwist.onLoanTo();
q->showBorrower();
ReturnBook(oliverTwist);
return 0;
}
```

```
//Listing 4
#include <iostream.h>
#include <string.h>

class Book;

class Author
        {
        public:
                Author(char* A);
                ~Author(){unlink(*this); delete
aName;}
                Book * Publication() {return b;}
                void showAuthor(){cout << aName <<
"\n";}
                friend void link (Author &a, Book
&b);
                friend void unlink(Author &a);
                friend void unlink (Book &b);
        private:
                Book *b;
                char* aName;
                void Publication(Book * bk){ b
=bk;}
        };

class Book
        {
        public:
                Book(char* T);
                ~Book()
                {unlink(*this); delete aTitle;}
                Author * writer() {return a;}
                void showBook()
                {cout << aTitle << "\n";}
```

```
	friend void link (Author &a, Book &b);
	friend void unlink(Author &a);
	friend void unlink (Book &b);
	private:
		Author *a;
		char* aTitle;
		void  writer( Author * auth )
		{a= auth;}
};

Author::Author(char* A)
	{
	b = 0;
	aName = new char[20];
	strcpy(aName, A);
	}

Book::Book(char* T)
{
a = 0;
aTitle = new char[20];
strcpy(aTitle, T);
	}


void link(Author & A, Book & B)
	{
	if(!A.Publication() && !B.writer())
		{
		A.Publication(&B);
		B.writer ( &A);
		}
	}

void unlink (Author & A)
	{
	if(A.Publication())
		{
		A.Publication()->writer(0);
		A.Publication(0);
		}
	}

void unlink (Book & B)
	{
	if( B.writer())
		{
		B.writer()->Publication(0);
		B.writer(0);
		}
	}

int  main()
{
Author dickens("Charles Dickens");
Book oliverTwist("Oliver Twist");
Book eyre("Jane Eyre");
Author bronte ("Charlotte Bronte");
Author wilde("Oscar Wilde");
Book pictureOfDorienGray("Picture of Dorien Gray");
link(dickens,oliverTwist);
link(wilde,pictureOfDorienGray);
link(bronte ,eyre);
cout << " Traverse from author to book\n";
cout << "Dickens wrote ";
dickens.Publication()->showBook();
cout << " Traverse from book to author\n";
cout << "Jane  Eyre was written by ";
	eyre.writer()->showAuthor();
unlink(dickens);
unlink(bronte );
unlink(wilde);
return 0;
}
```

```
//Listing 5
#include <string.h>
#include<iostream.h>
#include<cstring.h>
#include<classlib\assoc.h>
#include<classlib\dict.h>

class Tenant:string
{
public:
	Tenant():string(){};
	Tenant(char *A):string()
	{iname = new char[strlen(A) +1];
		strcpy(iname, A);}
	~Tenant(){delete iname;}
	int operator ==(const Tenant &T)const
	{return stricmp(iname, T.iname) ? 0:1;};
	void getTenant()const{cout << iname;}
	unsigned HashValue()const{return hash();}
private:
	char * iname;
};
class Apartment
{
public:
	Apartment(){};
	Apartment(char *D){pname = new
char[strlen(D)+1]; strcpy(pname, D);}
	~Apartment(){delete pname;}
	void getApartment()const {cout << pname;}
private:
	char *pname;
 };

typedef TIIAssociation<
Tenant,Apartment>Association;
typedef TIDictionaryAsHashTable<Association>
Lettings;
typedef
TIDictionaryAsHashTableIterator<Association>
			AssocIterator;

int main()
{
Lettings lettings;
Tenant * dave = new Tenant("Dave");
Apartment * hillside15a =
	new Apartment("15A Hillside");
Tenant * fred = new Tenant("Fred");
Apartment * hillside15b =
	new Apartment("15B Hillside");
Tenant * jane = new Tenant("Jane");
Apartment * hillside15c =
	new Apartment("15C Hillside");
Association *entry =
	new Association(dave, hillside15a);
lettings.Add(entry);
Association *entry1 =
	new Association(fred, hillside15b);
lettings.Add(entry1);
Association *entry2 =
	new Association(jane, hillside15c);
lettings.Add(entry2);
AssocIterator i(lettings);
Association *p;
const Apartment *flat;
const Tenant *occupier;
cout << "List lettings\n";
while(i)
	{
	p =i++;
	cout << "\t";
	flat = p->Value();
	flat->getApartment();
	cout << " has been let to ";
	occupier = p->Key();
	occupier->getTenant();
	cout << "\n";
	}
return 0;
}
```

```
// Listing6
#include <iostream.h>
#include <string.h>

class Tennant;
class Apartment;

class Lease
{
private:
	Tennant *i;
	Apartment *c;
	int rent;
	int duration;
public:
	Lease(Tennant& I, Apartment& C,
		int n, int d);
	~Lease();
	void getRent()
	{cout << "ú" << rent; cout << " PCM\n";}
	void getDuration()
	{cout <<  duration;cout << " months\n";}
	Apartment * flat(){return c;}
	Tennant *occupier(){return i;}
};

class Tennant
{
private:
	Lease *p;
	char * iname;
public:
	Tennant(char *A)
	{p = 0;iname = new char[strlen(A) +1];
		strcpy(iname, A);}
```

```
        ~Tennant(){delete p;delete iname;}
        void getTennant(){cout << iname <<"\n";}
        Lease * Lease(){return p;}
        Apartment *flat(){ return p ? p->flat()
:0;}
        friend class Lease;
};

class Apartment
{
private:
        Lease *p;
        char *pname;
public:
        Apartment(char *D)
        {p = 0;pname = new char[strlen(D)+1];
                strcpy(pname, D);}
        ~Apartment(){delete p;delete pname;}
        void getApartment(){cout << pname <<"\n";}
        Lease *Lease(){return p;}
        Tennant *occupier()
        {return p ? p->occupier():0;}
        friend class Lease;
};

Lease::Lease(Tennant &I,Apartment &C,int n,int d)
:i(&I),c(&C),rent(n),duration(d)
{
if(!i->p && !c->p)
        {
        i->p = this;
        c->p = this;
        }
else
        cout << "ERROR\n";
}

Lease::~Lease()
{
i->p = 0;
c->p = 0;
}

int main()
{
        Tennant * dave = new Tennant("Dave");
        Apartment * hillside15a =
                 new Apartment("15A Hillside");
        new Lease(*dave, *hillside15a, 234, 23);
        cout << "Where does Dave live? ";
        dave->flat()->getApartment();
        cout << "Who has tennancy of 15A Hillside?
";
        hillside15a->occupier()->getTennant();
        cout << "What is the rent of Daves' flat?
";
        dave->Lease()->getRent();
        cout << "How long is the lease on 15A "
                "Hillside? " ;
        hillside15a->Lease()->getDuration();
        delete dave;
        delete hillside15a;
        return 0;
}
```

```
//Listing 7
#include <iostream.h>
#include <string.h>

class Book;

class Borrower
        {
        public:
                Borrower(){};
                Borrower(char* A);
                ~Borrower()
                { delete [] bookLoan;delete
aName;}
                Book * Publication()
                {return *bookLoan;}
                void getBorrower()
                {cout << aName << "\n";}
                void ListBooks();
        friend void BorrowBook(Borrower &A, Book
&B);
        friend void ReturnBook( Book & B);
        private:
                Book ** bookLoan;
                char* aName;
                int booksOnLoan;
        };

class Book
        {
        public:
                Book(){};
                Book(char* T)
                {  aTitle = new char[20];
                        strcpy(aTitle, T);}
                ~Book(){delete aTitle;}
                Borrower * aquirer() {return a;}
                char * getBook(){return aTitle;}
                void WhoHas();
        friend void BorrowBook(Borrower &a, Book
&b);
        friend void ReturnBook( Book & B);
        private:
                Borrower *a;
                char *aTitle;
                void  aquirer( Borrower * br )
                {a = br;}

        };

Borrower::Borrower(char* A)
        {
        aName = new char[20];
        strcpy(aName, A);
        bookLoan = new Book *[5];
        booksOnLoan = 0;
        int i = 0;
        while (i < 5)
                bookLoan[i++]= NULL;
        }

void Borrower::ListBooks()
        {
        int x =0;
        while (x < 5)
        {
                if (bookLoan[x] != NULL)
                        cout <<"\t" <<
bookLoan[x]->getBook() << "\n";
                x++;
        }
}

void BorrowBook(Borrower & A, Book & B)
        {
        int i =0;
         while(A.bookLoan[i] !=NULL)
                        ++i;
        A. bookLoan[i] = & B;
        B.aquirer ( &A);
        A.booksOnLoan++;
        }

void ReturnBook ( Book & B)
{
Borrower * A = B.aquirer();
if(A->Publication())
{
        int i = A->booksOnLoan;
        while (i-->0)// A.booksOnLoan)
        if (A->bookLoan[i]->getBook() ==
                 B.aTitle)
                {
                        cout << "Book " <<
                                B.aTitle <<
                                "  returned by ";
                                A->getBorrower();
                        A->bookLoan[i] = 0;
                        A->booksOnLoan--;
                }
        A->Publication()->aquirer(0);
}
}

void Book::WhoHas()
{
        Borrower * br = aquirer();
        br->getBorrower();
}

int  main()
{
        Book eightyFour("1984");
        Book solentShores("Solent Shores");
        Book maidenVoyage("Maiden Voyage");
        Book vanityFair("Vanity Fair");
        Book sealord("Sealord");
        Book treasureIsland("Treasure Island");
        Borrower james("James");
        Borrower stevenson("Stevenson");
        BorrowBook(james,sealord);
        BorrowBook(james,vanityFair);
        BorrowBook(stevenson,solentShores);
        BorrowBook(stevenson,eightyFour);
        BorrowBook(stevenson,maidenVoyage);
        BorrowBook(stevenson,treasureIsland);
        cout << "List books borrowed by James\n";
        james.ListBooks();
        cout << "List books borrowed by
Stevenson\n";
        stevenson.ListBooks();
```

```
        cout << "Who has Maiden Voyage? ";
        maidenVoyage.WhoHas();
        cout << "Who has Sealord?  ";
        sealord.WhoHas();
        ReturnBook(sealord);
        ReturnBook(maidenVoyage);
        cout << "List books borrowed by James\n";
        james.ListBooks();
        cout << "List books borrowed by
Stevenson\n";
        stevenson.ListBooks();
        return 0;
}
```

```
// listing 8
#include <iostream.h>
#include <string.h>

class Actor;
class TVSeries;

class node
{
private:
friend class BList;
node *next;
void * pd;
};

class BList
{
public:
        BList(){start = 0;}
        ~BList();
        void insert(void *p);
        void *remove();
        void *remove(void *p);
        void reset();
        void *next();
private:
        node *start;
        node *c;
};

BList::~BList()
{
node *p1, *p2;
if(!start) return;
p1 =start;
while (p1)
        {
        p2 = p1->next;
        delete p1;
        p1 = p2;
        }
}

void BList:: insert(void *p)
{
node *temp;
temp = new node;
if(!start)
        {
        start = temp;
        temp->next = 0;
        }
else
        {
        temp->next = start;
        start = temp;
        }
temp -> pd  = p;
}


void * BList::remove()
{
while(start)
        {
        node * p1 = start;
        start = start->next;
        delete  p1;
        }
return 0;
}



void * BList::remove(void * p)
{
node *p1, *p2;
p1 = start;
p2 = 0;
if(p1->pd == p)
        {
        p2 = p1->next;
        delete p1;
        start = p2;
        }
else
        {
        while(p1)
                {
                p2 =p1->next;
                if(p2->pd == p)
                        {
                        p1->next = p2->next;
                        delete p2;
                        return p;
                        }
                p1 = p1->next;
                }
      }
return 0;
}



void  BList::reset()
{
c = start;
}

void * BList::next()
{
if(c)
        {
        void *r = c->pd;
        c= c->next;
        return r;
        }
else
        return 0;
}

class CList
        {
        private:
                BList v;
                CList(const CList &);
                CList & operator =(const CList &);
        public:
                CList() : v() {};
                ~CList(){};
                void reset(){v.reset();}
                Actor * remove()
                {return(Actor *)v.remove();}
                Actor * remove(Actor *p)
                {return(Actor *)v.remove(p);}
                void insert(Actor
*p){v.insert(p);}
                Actor * next()
                {return(Actor *)v.next();}
                };

 class RList
        {
        private:
                BList v;
                RList(const RList &);
                RList & operator =(const RList &);
        public:
                RList() : v() {};
                ~RList(){};
                void reset(){v.reset();}
                TVSeries * remove()
                {return(TVSeries *)v.remove();}
                TVSeries * remove(TVSeries *p)
                {return(TVSeries *)v.remove(p);}
                void insert(TVSeries *p)
                {v.insert(p);}
                TVSeries * next()
                {return(TVSeries *)v.next();}
         };

class TVSeries
        {
        private:
                CList cast;
                char* aName;
                void insert(Actor * t)
                {cast.insert(t);}
                Actor * remove()
                {return cast.remove();}
                Actor * remove(Actor *A)
                {return cast.remove(A);}
        public:
                TVSeries():cast(){};
                TVSeries(char* A):cast()
                {aName = new char[20];
                        strcpy(aName, A);}
                ~TVSeries()
                {delete aName; cast.remove();}
                Actor * Appearance()
                {return cast.next();}
                void getTVSeries()
```

```
                 {cout << "\t" << "TV Series:\t"
                        << aName << "\n";}
                 void reset(){cast.reset();}
                 void ListCast();
        friend void link (TVSeries &A, Actor &B);
        friend void unlink( TVSeries & A, Actor &
B);
     };

class Actor
        {
        private:
                 RList roles;
                 char* aName;
                 void insert(TVSeries * t)
                 {roles.insert(t);}
                 TVSeries * remove()
                 {return roles.remove();}
                 TVSeries * remove(TVSeries *A)
                 {return roles.remove(A);}
        public:
                 Actor():roles(){};
                 Actor(char* A):roles()
                 {aName = new char[20];
                        strcpy(aName, A);}
                 ~Actor()
                 {delete aName; roles.remove();}
                 TVSeries * CastOfActors()
                 {return roles.next();}
                 void getActor()
                 {cout << "\t" << "Actor:\t"
                        << aName << "\n";}
                 void reset(){roles.reset();}
                 void ListAppearances();
        friend void link (TVSeries &A, Actor &B);
        friend void unlink( TVSeries & T, Actor &
A);
        };

void TVSeries::ListCast()
                 {
        Actor * p ;
        reset();
        while ((p=Appearance())!=0)
                 p->getActor();
        }

void Actor::ListAppearances()
        {
        TVSeries * p ;
        reset();
        while ((p=CastOfActors())!=0)
                 p->getTVSeries();
        }

void link(TVSeries & A, Actor & B)
        {
        B.insert(&A);
        A.insert(&B);
        }

void unlink ( TVSeries & A, Actor & B)
        {
        B.remove(&A);
        A.remove(&B);
        }

int main()
        {
        //Create TV show and actor objects
        TVSeries minder("Minder");
        TVSeries sweeney("The Sweeney");
        TVSeries morse("Inspector Morse");
        Actor cole("George Cole");
        Actor waterman("Dennis Waterman");
        Actor thaw("John Thaw");
        Actor foster("Barry Foster");
        link (minder, waterman);
        link (sweeney, waterman);
        link (minder, cole);
        link (sweeney, thaw);
        link (sweeney, foster);
        cout << "List the cast of 'The Sweeney'\n";
        sweeney.ListCast();
        cout << "List the cast of 'Minder'\n";
        minder.ListCast();
        cout << "Dennis Waterman apeared in\n";
        waterman.ListAppearances();
        cout << "Unlink sweeney, waterman\n";
        unlink(sweeney, waterman);
        cout << "Dennis Waterman appeared in:\n";
        waterman.ListAppearances();
        cout << "List the cast of 'The Sweeney'\n";
        sweeney.ListCast();
        return 0;
        }
```

Java? Where is that?

by The Harpist

One of the hottest topics on the Internet these days is a new language from Sun Microsystems called Java. The purpose of this article is to introduce you, as a C++ user, to Java. Some may see it as competition, I do not. I think we should welcome it and provide information to ACCU members as well as invite Java users to join us. In the long run, *Overload* may not be the right place for Java topics, maybe it will be entitled to a publication of its own – though that will have to depend on interest as well as an editor becoming available.

Java joins several other object-oriented derivatives of C. The best known is C++ and the most tenuously connected is Eiffel. There is also Objective C – the development of Tom Love and best known in the NextStep environment. Two important features contribute to the success of a language. The first is that it is easy to learn, based on prior knowledge. The second is an almost immeasurable quality of 'timeliness'.

Both Eiffel and Objective C are easy languages to learn if you are familiar with C syntax. The thing that has inhibited the move to Eiffel is the need to radically change one's programming style (or what the Americans call a paradigm). Combining this with an early shortage of compilers, the relative slowness of those that did exist and natural human resistance to changing to something because 'it was better for you' resulted in the initial uptake of Eiffel being slow. Many programmers resent being told that they should change languages because the new one will prevent them from doing silly things.

I'd say Eiffel owes more to Pascal than C but that's basically irrelevant to The Harpist's point – Ed.

Objective C probably suffered from being promoted with an excellent operating system that in turn was bound to a specific piece of hardware. By the time all the bits had been decoupled in peoples minds, C++ was up and running.

So why did C++ succeed. Probably three major elements brought this about. Initially it was something that was evolving from C. Those using "C with Classes" did not immediately realise that their new tools were going to radically change their way of working. The most obvious

extensions just met problems that they were having with C. The second item was the rapid development of Cfront which made C++ accessible to anyone with a suitable C compiler. The overwhelmingly important element was that C++ was being developed by a major user of software and software development tools. C++ spread very rapidly through the telecommunications industry. That C was the native language of Unix made it even easier for C++ to spread.

Francis Glassborow takes every opportunity to highlight the fact that evolution eventually produces different species. He believes that C and C++ are now quite different languages, While there is some truth in this view, it is not sufficient to force the two programming cultures apart. Many programmers need to use both languages. C++ is inextricably bound to the mistakes in C – rather like the problem Intel has with the binding between the Pentium and its ancestors, the 8088, the 8080 and the even earlier 4040. We know that early design decisions make little if any sense today but we are bound to maintain the existing interface (after all that is one of the fundamental pillars of object-orientation).

Now suppose that another vibrant culture arose in which there was no need to retain compatibility with the past, what language would you then design? One design criterion would be that it should be easy for those familiar with C and C++ (Objective C and even Eiffel) to learn. In other words it should have a similar ‘look and feel’. But if we were no longer constrained by the need to use and maintain legacy code we could free ourselves of many problems. I am not going to list them here, but any honest C/C++ programmer knows that there are a multitude of problems that make these languages tarpits for the inexperienced.

What else might we want from a language that was to be a kind of redesigned C with classes? Platform independence certainly must be a strong contender. This strongly suggests some form of virtual machine. Actually we have visited this problem with portability before, p-code and USCD Pascal. In the last twenty years considerable advances have been made, but something along the lines of p-code with a powerful modern interpreter would seem possible. If efficiency mattered we could use the kind of interpreter that converts the code to machine code as it is first executed – i.e., a slightly slower first pass through any piece of code but with much faster subsequent passes.

Support for distributed processing would also seem to be a good candidate for consideration. Throw in multi-threading and we begin to have something that would look attractive to modern network users.

Over the last four years there has been an explosive growth of Internet use. This has incorporated many substantial innovations. One of these is the concept of active documents. The latest multi-user games have full motion activity – for example you can now fly combat missions against other players. However much the bandwidth has been improving we are far from being able to send full motion, interactive graphics through telecommunications networks. What we can do is to send data to an interpreter at the other end. In effect, we are sending a kind of p-code to an interpreter. As players want to use their own favourite computing platform, the data (p-code) is platform independent, it is the interpreter’s job to convert it into visuals for the owner’s hardware.

Very little of this is new – what is new is that we have literally millions of people who are interested in using ‘active’ pages and a substantial proportion of those want to be able to do it for themselves. This provides an environment in which a new programming language can take root and flourish. Note that users will not be tied by legacy code, but many of them will already have some familiarity with either C or C++.

This is where Java comes in. I would counsel against being sucked in by all the hype that is flying around, but I would also advise you to find out about Java. Let me list a few points:

- It is designed by a single group from Sun Microsystems who are familiar with both C and C++.
- It is only currently available in various alpha’s and beta’s
- It requires (or seems to) a 32-bit (or larger) platform
- It requires a platform that can support multi-threading

Currently the only versions available are a beta for Solaris and a late alpha (admittedly with a couple of serious bugs) for Windows NT and

Windows 95. There are other versions being developed by groups outside Sun Microsystems.

There are substantial commercial interests in its success and widespread adoption.

Despite claims by enthusiasts, there are good reasons why systems administrators (and that includes you if you have your own Internet connection) should be wary of allowing HotJava (interpreters for code provided remotely) to run on their machine. How happy are you with the idea of allowing externally provided code to run on your hardware? As one of the things for which Java is being advocated is updating your software with patches, I must wonder about the potential for updating software with viruses. If remotely provided code can touch your disk, there are inherent dangers.

Java is very C++ like, but many of the things that cause the greatest problems have been replaced. For example multithreading is built into the language (the programmer does not have to handle potential race conditions etc).

Java uses garbage collection (running as a low priority thread) so the programmer no longer has to do the memory management.

Java does not use pointers so all the bugs those can produce have gone. Don't start jumping up and down and shouting about how useful pointers are: stop and think about whether you would prefer the same functionality without pointers? Java provides true array types, so this aspect of pointers has gone. References are the norm for passing objects around.

Built-in types are strictly and completely defined. No more of the problems we have with different implementations using different size **int**s and different rules for negative values.

Pascal programmers will be delighted to find that Java does not support any form of automatic conversions between built-in types. If you want to divide an **int** by a **float** and store the answer in a **long** you will need to make the conversions explicit.

Java is object-oriented. There are no procedural aspects (except within the context of a single class method). Like Smalltalk, all classes are derived from the single superclass 'object' and there is no multiple inheritance.

No multiple inheritance? Yuk! :-) – Ed.

I could go on but I think you should get the drift by now.

Java v C++

I would be profoundly unhappy if someone suggested that Java should replace C++. However, remember that C++ claims that its great strength is in writing programs that are fifty-thousand lines plus. I have absolutely no doubt that C++ should and will remain a vitally important computer language. Having said that let me suggest some areas where Java might be a powerful alternative.

The first is the ordinary hobbyist programmers. These will find Java an easier language to use without constantly shooting oneself in the foot. It will also provide portability – that carefully crafted demo of one's programming skills will run on your friends' machines.

Next we have the professional programmer who needs to develop small special purpose programs. Again, the advantage of platform independence coupled with Java's features supporting robustness and security will prove attractive.

What about those that want to write material to run over a LAN? I think they will find that Java has a lot to offer.

Those working in application areas where multithreading or garbage collection are advantageous will find that Java is a strong candidate for their work.

Of course there will be a heavy push from those surfing the net to adopt Java for their activities. These, I think, will provide the first impetus for wide availability of Java interpreters and development systems.

Conclusion

Just as I do not believe C++ replaces C, I do not think that Java need replace either. The advantage of having a third language with very similar syntax is that experienced programmers will be able to capitalise on their skills while choosing the most appropriate tools for the current task. C is ideal where efficient value based programming is essential. C++ has great strengths where mixed paradigm programming is important. It also has considerable advantages for large scale programming. Java would seem to be a good candidate for object-oriented, multi-threaded programming.

This is only a brief first view of a new language derived from C. I hope that many readers will have open minds, try Java for themselves and feed back their experiences to ACCU members via an appropriate publication.

The ball is now in your court.

The Harpist

Addendum from Francis Glassborow

Some people are puzzled by the names HotJava and Java. HotJava is a WWW browser from Sun Microsystems that can be extended to handle a variety of protocols (ftp, http, mail servers etc) via applets written in Java. Java is the programming language used to write these applets. Java is a full programming langauge and can be used (as The Harpist indicates) to do ordinary OO programming.

Francis Glassborow
francis@robinton.demon.co.uk

If there is sufficient interest, I am happy to run a "Java Corner" in Overload based on your contributions – 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!

In addition to my regular column on the progress of the standard, Francis Glassborow reflects on the meaning of linkage and Kevlin Henney considers a couple of possible language changes that might make C++ more consistent.

The Casting Vote

by Sean A. Corfield

The location of the latest meeting was exotic enough to make up for the relatively dull decisions made: Tokyo was the venue for the November '95 ISO/ANSI C++ meeting.

Not that we should be doing anything exciting at this stage of the standards process – the focus of the committee's work is on resolving small issues now. A large number of these small problems were sorted out with the committee voting on 31 motions, all of which dealt with one or more "bug" in the draft.

Debugging the draft

I said "bug" because in many ways the C++ standards process can be viewed like any other software project: we have a tight deadline and limited resources and we have to release a product onto the world market for which there will effectively be no upgrades for many years. We produced an "alpha" release this year – the first CD – and several countries rejected it, including France, Germany, Netherlands, Sweden and the UK. At the moment, we are fixing the problems identified by those alpha testers so that we can ship a beta release next year – the second CD. If that proves acceptable, we can ship the prerelease version in 1997 – the DIS – which hopefully will require no more than a few typos being fixed.

So how do you manage bug reports on 700+ pages of "source"? Each module – "clause" – is the responsibility of one member of the committee who gathers bug reports for their module, analyses the problems and suggests possible resolutions. A subgroup of the committee then examines and sometimes reworks the resolution to produce a proposal for the full committee. If the majority of the committee think the fix will "work", it is accepted and the "programmers" – the Project Editor[1] and his team of helpers – integrate the resolution into the source.

Testing the fixes

This is the hard part at the moment: there are no compilers that implement the whole draft. Without widespread support for, and use of, the language features that we are "fixing" we cannot test the solutions to any great extent. Microsoft's latest offering supports **namespace** and RTTI, alongside templates and exception handling.

[1] I intend no slight to Andrew Koenig by the classification of "programmer" here! The Project Editor's job carries a tremendous amount of responsibility and is damned hard work – I don't believe there is a true simile within the software development world.

Soon, many UNIX vendors will also be offering **namespace** and we will begin to see how this feature behaves in use and whether the small fixes applied at this meeting were, in fact, correct – I believe they were.

The same is true for other parts of the standard and especially so for the library. Although implementations of parts of the library have been available for some time, I find it hard to accept that the gothic monstrousities that are *locale* and *iostream* have been seriously tested in commercial use.

We have to face the fact that when C++ v1.0 appears, it may have many dark untested corners and v1.1 will be some years away – v2.0 will be 5 to 10 years away.

On the horizon

Perhaps more interesting than what we did at this meeting is what we have yet to solve. The library has a myriad open issues – some of you who've been trying to use *auto_ptr* or the STL will have already encountered some of them. Within the language itself, there are probably only three or four dozen known bugs that we have to solve next time. However, two of those are long-standing, difficult problems:

- name injection
- template compilation

We now have ideas on how to resolve these and solutions should be available for the next committee meeting – which will be reported on in *Overload 13* (April '96). That's all I have time for – I have some **template** papers to write for the post-Tokyo mailing!

Sean A. Corfield
Object Consultancy Services
ocs@corf.demon.co.uk

Some thoughts on linkage

by Francis Glassborow

I have just recently come to understand what linkage is about in C and C++ and thought I would share my insights with the rest of you (and the experts can tell me where I am wrong).

I suppose most of us, who think about it, think that linkage is something to do with the linker. That then leaves us slightly mystified by C's internal and external linkage. If linkage were something to do with the linker, what is the significance of internal linkage as opposed to no linkage. C seems to say that internal linkage is some sort of linkage that explicitly is not the concern of the linker. So let me take you back to the drawing board because this whole issue is very important in C++ while being largely trivial to the C programmer.

Linkage is about the declaration of names (i.e., the process of giving an identifier a meaning). It is not directly connected to definitions except that we cannot actually define something without also declaring its identifier – that is the way languages derived from C work. We can declare without defining but we cannot define without declaring (actually K&R C complicated thing by talking about tentative definitions, or is it tentative declarations? I am really sublimely uninterested because I think it is just another place where poor choice of terminology only serves to confuse).

Now C is quite clear: you can only define something once. If you try to do it twice, either the compiler (if it is in the same file) or the linker (if it is in different files) will spot it and stop you. C++ cannot work with this simple view, things like class definitions, **inline** functions (yes I know C++ uses a hack for this one and the next one, but I will get round to that) and **const** 'globals' may need to have definitions in all the files where they are used. Remember that the concept of a file is largely a human artifact though the idea of compiling in sections is valuable. This is why C++ has had to come up with what is called the 'One Definition Rule'. This enshrines the intent that however many times some items must appear to be defined in a program, it must behave as if there is only a single definition.

Now some identifiers can only be declared once: there is no legal way of declaring them twice. Parameters of functions are like this – they turn up in the definition (their appearance in a prototype is something else and has no connection with the names used in the definition). Such names are said to have *no linkage*. If such names are declared two or more times, it is either an error or the scope is different (and so they only look the same).

Mostly, identifiers can be declared more than once. For example, the name of a function can be both declared (as a prototype) and defined (in either the same file or another one). Names that can be declared more than once require linkage,

that is, they have a quality that can be used at some stage to connect the multiple instances to a single ‘thing’. Ideally, that is all we need. Some names are declared in contexts that mean they cannot be redeclared and are said to have no linkage. Other names are declared in contexts where there may sometimes be another declaration of the same name elsewhere (like function names as opposed to parameters) – these must possess the property of *linkage*.

Up to here, life is simple. However, C actually needs redeclaration of the same name in a single file even if the name is not supposed to leak out of a single file context. For example, when two structs contain pointers to each other (mutual recursion), one must be declared before it is defined:

```
struct A;    /* just a declaration */
struct B {   /* a combined declaration
                and definition */
  struct A* pntA;
  /* other members */
};
struct A {   /* now its definition that
                requires linking to the
                earlier declaration */
  struct B* pntB;
  /* other members */
};
```

If you need this kind of data structure you must have linkage. If you do not want the name to leak out you must do something else as well. What C did was to invent two flavours of linkage, internal (basically for the benefit of the compiler) and external (largely for the linker). By default the names of functions have external linkage and we have to take specific action to restrict the linkage. Remember that C hated adding keywords, so it abused one that already existed and instead of having **intern** it used **static**. Silly, but it no doubt seemed a good idea at the time.

In C, types always have no linkage but in C++ they do indeed have linkage – just to further confuse the issue – Ed.

The problem with variables is slightly different. The default declaration for a variable is a definition as well, so in this case we have two problems to cope with. When a global variable is intended for use in more than one file we have to stop the extra declarations from being redefinitions and causing havoc with the dumb linker technology of the seventies. So now we introduce the keyword **extern** which doesn’t actually mean what it appears to mean. What it means in the context of a global variable is that you are only declaring the name, *not* defining it (the definition is elsewhere). Global variables have external linkage by default, just like functions and if we want them to be restricted to a single file (i.e., have internal linkage) we have got to use that **static** keyword again. Of course **static** has a perfectly valid meaning in the context of a local variable definition where it means that the variable must be placed in static memory. It may be very nice to keep to not more than 32 keywords when you want to pretend that you have some virtual C machine that wants to store the keyword tokens in 5-bits, but accepting a limit of 64 keywords would have made life much easier.

Before you all start writing in about that last sentence, I’ll comment that I am assuming it is Francis having a little joke! – Ed.

Now let me move to C++. In the early days C++ simply accepted the C concept of two flavours of linkage. But as time has passed this has looked increasingly artificial, as well as producing some quite unpleasant results. For example I recently wanted to use *__FILE__* as a non-type argument to a template.

The rules, as they currently stand, say that non-type template arguments must either be a constant of builtin type or be addresses of objects that have external linkage (I suspect that this is related to the way Cfront implemented templates). Now let me consider my options:

```
template <char[] ac> class Ta ...
```

won’t work because arrays will not do. This may seem a bit of language awkwardness, but if you think about it you will realise that allowing it would make life difficult. So it looks as if it will have to be:

```
template <char* pc> class Tc ...
```

So now lets move to the point of declaration of an instance:

```
Tc<__FILE__>
```

won’t cut it. Remember that *__FILE__* is a preprocessor macro so by the time the compiler sees your code you will be trying to instantiate a template with a literal string.

So the next shot might look something like this:

```
char* file = __FILE__;
Tc<file> filename;
```

Fine until you declare (define) *file* in a second source file and you get clobbered by the linker

for redefinition. It's no good using **extern**, because we want the correct string in the context of each file. So the 'obvious' step is to write:

```
static char* file = __FILE__;
Tc<file> filename;
```

This actually works with Borland C++ but breaks the current rules because the non-type argument is required to have external linkage. I have no doubt that there are workable alternatives but my point is that we are suffering from an historical separation of linkage into two flavours (I think it was a hack). We simply will not need this distinction in C++ once **namespace** is generally available. Any name that we want to keep within the scope of a file (translation unit if you want to be technical) can be placed in the unnamed namespace. Such names will have external linkage (i.e., the linker etc. can see it) but will have a unique 'unpronouncable' namespace qualification that will distinguish it from all apparently similar names used in other files. This may sound complicated, but it simply means that C++ now has a mechanism for stopping names leaking out of files and so we no longer need two flavours of linkage. All names can either have linkage (with the name qualified by its namespace), global, unnamed (and hence secretly provided with a unique qualification shared by names in the unnamed namespace in a single file) or named; or have no linkage because they are inherently not redeclarable.

For this to work, we need to make all the file-scope uses of **static** (both explicit and implicit, e.g., **const int** *j = 7***;** at file scope) be synonyms for declaration in the unnamed namespace. Such a change should simplify things for everyone.

The idea of an unnamed (secretly named) scope is so useful that I would not be surprised to see C adopt it in its next revision. The only apparent problem in implementing a secret name is that it makes true names longer. But if you, as a programmer, cannot utter the hidden name the compiler could always just not export the names in a C context (that is, do what it does now). If true names are required to support template technology in C++ then the unutterable secret name solves the problem. Neither C nor C++ needs to have two flavours of linkage so let us quietly bury them. As most programmers do not understand linkage anyway, this would be a benefit to many.

The floor is yours.

Francis Glassborow
francis@robinton.demon.co.uk

Literally yours
by Kevlin Henney

The actual type of a string literal is an artefact from C that continues to plague C++. When seen in the initialiser for an array of **char** it acts as a short hand form of an aggregate initialiser, but in other expressions it is a pointer to a string of static storage duration. More accurately we might say that a string literal refers to an unnameable array declared static within a declaration unit:

```
void foo(char *bar)
{
  strcpy(bar, "foo");
}
```

can be considered equivalent to

```
static char __str_0001__[4] =
                    {'f', 'o', 'o',
'\0'};
void foo(char *bar)
{
  strcpy(bar, __str_0001__);
}
```

Where *__str_0001__* represents a compiler generated name for the dummy array that represents the literal. This array view of string literals gives the correct answer for **sizeof**, i.e., **sizeof** "a literal string" gives 17 rather than **sizeof(char*)**.

Intuitively, however, there is something wrong here: literals are normally considered to be manifest constants. Originally C had no way of declaring objects as constant, so there was simply no way of expressing that a string literal was an immutable array of **char**. With ANSI C came the introduction of **const** – borrowed, in fact, from the youthful C++. Also with ANSI C came the requirement for backward compatibility. This prevented the obvious and desirable move of defining string literals as **const**, by taking into account the important and even more desirable aim of not breaking almost every K&R C program ever written.

Instead, the committee contented themselves with saying that the type of a string literal was not **const**, but any attempt to modify it would result in undefined behaviour. This leaves implementations free to put string literals in write-protected memory. In effect, a string literal is **const** but its compilation type is not, i.e., this is a part of C's typing expressed outside of the type

system. It is as if the implementation were equivalent to

```
static const char __str_0001__[4] =
                        {'f', 'o', 'o',
'\0'};
void foo(char *bar)
{
  strcpy(bar, (char *)__str_0001__);
}
```

In C, we simply learn to discipline ourselves and ensure that we, as programmers, only attach string literals to **const char***:

```
char *do_not_do_this = "bad practice";
const char *do_this_instead =
                        "good practice";
```

We can get by with this in C, and some code checking tools will give us a hand in shoring up the type system. With C++ many of the same issues remain, but there is an added complexity: overloading. Whereas C allowed us to transparently treat string literals as **const**, although the compiler front end considered them to be otherwise, C++ defeats us with its dexterity and convenience:

```
void call_me(char* modifiable)
{
  reverse(modifiable,
          modifiable +
strlen(modifiable));
  cout << "modified: " << modifiable
       << endl;
}

void call_me(const char* unmodifiable)
{
  cout << "unmodified: "
       << unmodifiable << endl;
}
...
call_me("kenneth"); // void
call_me(char*)
```

In other words, the overloading is counter-intuitive and we have no language support for our expectations – not a better C. A solution to this would be to consider a string literal **const** for all matching purposes. In the absence of a good match a compiler could then consider an implicit **const_cast** ahead of the literal. This implicit conversion to non-const could be marked up as deprecated (highlighted for potential removal from a future standard) and would elicit a diagnostic from good compilers. The same rules would apply to wide character string literals.

Given that the joint standardisation committees have thrown out the default **int** rule, there is no reason that they should not at least try to fix string literals for future generations. And fixing is indeed the correct word: they can only be considered broken as they stand. Sean made a proposal along these lines a while back, but it was not accepted: the core working group was having a day of indifference at the time. This was obviously not the same day the ludicrous proposal was accepted for *main*'s return value to default to 0 if the programmer forgot or was too lazy to put in a **return** statement. It is the compiler's job to diagnose, not to correct, broken programs.

I intend to raise this issue again at the next BSI C++ panel meeting, and would be interested in reader opinion (either directly to myself or via Sean). Come to that, what do you think of *main*'s default return value – inspiration or desperation?

Kevlin Henney
kevlin@two-sdg.demon.co.uk

I think the sanctioning of such sloppiness was a dreadful idea – the argument in favour was that main is a very special function, despite the fact that it looks just like any other C++ function. There are moves afoot to exempt main from the default int rule too – something I shall strongly oppose! – Ed.

Anonymously yours

by Kevlin Henney

What is to **struct** as

```
union
{
  long   as_long;
  double as_double;
};
```

is to **union**? The answer is, at the moment, nothing.

Unions invite danger, but they have their uses. Although less common in C++ than in C – derivation covering many of the uses – C++ offers a method to wrap them up behind a safe class interface (see Stroustrup's *The C++ Programming Language, second edition*, for an example).

What anonymous unions offer is a simple flattening of name space, i.e., I do not have to refer to the union by name followed by one of its members, as the union member names are considered

to be in the enclosing scope. Anonymous unions may be used within other structures, in local scope, or at file scope with internal linkage. To all intents and purposes the members simply become variables in the relevant scope, albeit at the same offset:

```
if (convert(text, as_long)) ...
else if (convert(text, as_double)) ...
else ...
```

So what would anonymous structs offer? At first sight they may seem a little redundant: separately named variables at separate offsets in the enclosing name space – why not just declare them as separate variables? Consider, however, the following:

```
int first;
int second;
...
if (&first < &second) ...
```

Is the result defined? No. But the following would be:

```
struct
{
  int first;
  int second;
};
...
if (&first < &second) ...
```

Members in an aggregate declared within the same access group are contiguous with ascending addresses, so such comparisons are well defined. A feature like this is redundant within an enclosing struct, but at file scope and in local scope an ordering would now be present where there was previously none. For instance, a number of dummy opaque types could be defined statically within a translation unit and at once be named and ordered.

Flattening the member access path also finds use within anonymous unions:

```
class number
{
  ...
private:
  ...
  union
  {
    double real;
    struct
    {
      int numerator;
      int denominator;
    };
  };
};
```

By imposing an ordering on members, anonymous **struct**s have a role to play with regards to systems programming, convenience, and orthogonality. However, they are not currently defined in C++. Should they be?

Kevlin Henney
kevlin@two-sdg.demon.co.uk

C++ Techniques

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

Roger Lever continues his series on writing useful classes for debugging, Uli Breymann looks at the difficulties involved in writing a correct assignment operator, Kevlin Henney revisits the *Address* class problem and also begins a new series on template techniques which I hope will open your eyes and expand your mind!

Simple classes for debugging in C++ – Part 2

by Roger Lever

Part 1 laid the foundations of a simple debugging class. This was done by using a minimal inheritance hierarchy of *Base* and *Derived* combined with a test *main*() which showed object construction and destruction, along with a few common problems. *RNLI* was introduced as the debugging class via inheritance, but it is currently very basic, only outputting its creation and destruction events. Some design decisions associated with what *RNLI* would do were also discussed.

The overall gameplan for RNLI's growth was outlined in Part 1:

- Very basic debug class which will output state messages
- Provide some macro magic to automatically enable or disable debug
- Differentiating between memory allocated statically and with **new**

- Provide some heap walking capability to "see" what's in memory
- Output debugging information to a file

Macro magic

C already has a well established and well known mechanism for placing debug code within the source file(s) which can be easily removed. The mechanism is to use macros, statements that the compiler will expand (or contract to nothing) during the precompilation phase. This expansion or contraction can be controlled by a flag variable which is used during precompilation. The ANSI C library's *assert* is an example of this, where the assert statement is included into the final program based on whether NDEBUG is defined. C++ has inherited these macro tricks and it will be used here to insert or remove *RNLI* from the code. The code to do this is split into three parts:

1) Flag status variable, either defined (in *main.cpp*) or not
   #define *CHECK_ON*
   Note that *CHECK_ON* must be defined *before* the **#include** of *rnli.h*
2) Macros to either use *RNLI* via inheritance, or nothing
   #ifdef *CHECK_ON*
   #define *USE_CHECK* **: public** *RNLI*
   #else
   #define *USE_CHECK*
   #endif
3) Use a disciplined approach to class declarations
   class MyClass *USE_CHECK*
   {
   // *whatever*
   };

This process auto-magically includes or removes *RNLI* from the class declaration depending on whether *CHECK_ON* has been defined or not:

```
class MyClass
  : public RNLI {}; // CHECK_ON defined
class MyClass {};   // CHECK_ON not
defined
```

This establishes the basic mechanism for adding *RNLI* to classes and removing it again from the final production code.

Canonical form for RNLI

Various authors of C++ books talk of a canonical class declaration or what every class should contain. Cline [1] uses the term "The Big Three" to refer to members that will be created by the compiler by default if the class declaration does not include them:

- Destructor
- Assignment operator
- Copy constructor

Cline omits the default constructor, which will also be created by the compiler. Coplien [2] has stated a more complete list, that for any class *X*, it should contain:

- A default constructor (*X*::*X*())
- A copy constructor (*X*::*X*(**const** *X***&**))
- An assignment operator
  (*X***&** *X***::operator=(const** *X***&**))
- A destructor (*X*::~I())

The guiding principle here is to avoid surprises from the compiler which will generate these declarations and definitions, if you the programmer do not, and the need arises. Of course, if you *know* that the compiler will never need to generate one of these or that a compiler generated default one is fine for the task – then it can be omitted. However, can you be that sure? Even if you do know *now*, will you know in six months time? Will your successor? Leave that visual reminder, declare the class in the canonical form.

There is another important benefit from these visual reminders, they have also declared the original designer's intent: that *RNLI* is not designed to support the copy construction and assignment operations. It has formed part of the class documentation at the point it is most needed (or most likely to be read) – the source. So, *RNLI* becomes:

```
class RNLI {
public:
  RNLI()
  { cout << "RNLI constructor\n"; }
  virtual ~RNLI()
  { cout << "RNLI destructor\n"; }
private:
  RNLI& operator=(const RNLI& r);
  RNLI(const RNLI& r);
};
```

The assignment operator and copy constructor are declared **private** to disable expressions like:

```
RNLI r1;
RNLI r2 = r1;
```

By declaring them **private** no-one else has access to them. They do not need to be defined

within *rnli.cpp*, the compiler will not issue warnings or error messages. *RNLI* has a virtual destructor – why?

Virtual destructor

The virtual destructor enables the runtime mechanism to call the appropriate destruction routine of the object. This process ensures that all of the allocated objects are destroyed correctly. If, for example, *RNLI* was declared with an ordinary destructor it would be very easy to engineer an accident:

```
// Original code and output for
comparison
Base* ptrD = new Derived;
ptrD->print();
delete ptrD;

RNLI contructor
Base constructor
Derived constructor
Derived print
Derived destructor
Base destructor
RNLI destructor

// Modified version and output
RNLI* ptrD = new Derived;
delete ptrD;

RNLI contructor
Base constructor
Derived constructor
RNLI destructor
```

Clearly the *Base* and *Derived* components have not been destroyed. The fact that both *Base* and *Derived* do have **virtual** destructors is irrelevant since the runtime mechanism will never get beyond *RNLI*. Contrived? Yes, but not much. It is a very simple mistake to make and possibly could go unnoticed for quite a while. On the other hand, if *Derived* controlled a vital resource such as an important file or a semaphore it would be much more noticeable!

Building up RNLI's interface

RNLI is pretty useless right now. It has not provided any services or behaviour that will help in a debugging process. What is needed now is to add some real functionality to:

a) Track the creation of *RNLI* derived objects

b) Track the destruction of *RNLI* derived objects

c) Provide a mechanism to validate individual objects

A straightforward way to provide this set of services is to create a collection of *RNLI* objects. In C++ each object has an implicit **this** member, *RNLI* could use this during construction to identify a particular object and maintain it within its collection. Consequently, the original object could be verified by checking it against the corresponding *RNLI* collection item. This is not a totally foolproof mechanism but it is adequate for the purpose. By adding this collection capability *RNLI* will start to gain some substance:

```
class RNLI {
public: // as before plus...
  bool isValid() const
  { return (this == me); }

private: // as before plus...
  RNLI* me;
  RNLI* next;
  static RNLI* rnliHeap;
};
```

The *me* and *next* member variables are used to track objects during their construction by adding them to a static (shared between *RNLI*s) list – *rnliHeap*. The reason for naming the list *rnliHeap* will become clear when we add another static list to the class later.

Now that an object is identified with *me* and is held in a simple list, the original object can be verified using *isValid*(). This member function will compare the object's **this** member with *me* and return the result.

Building up RNLI's implementation

The implementation details for these addtional members are:

```
RNLI* RNLI::rnliHeap = 0;
```

The construction and destruction event will still be signalled by a message, however, the important detail of tracking the object is now being done.

```
RNLI::RNLI()
: me(this) {
  next = rnliHeap;
  rnliHeap = this;
  cout << "RNLI constructor" << endl;
}

RNLI::~RNLI(){
  assert(isValid());

  rnliHeap = me->next;
  cout << "RNLI destructor" << endl;

  me = 0;
  next = 0;
}
```

The destructor includes a validity check prior to its action helping to ensure that the object being destroyed is still valid. The *RNLI* pointers (*me* and *next*) are set to zero for sanity checking.

RNLI test run

The redefinition of *RNLI* so far has not changed the output of the test program, what has been done with *RNLI* is transparent to *Base* and *Derived* and the contents of *main.cpp*. It would be interesting to see what happens now if a buglet is introduced:

```
Base* ptrD = new Derived;
Base* ptrD1 = new Derived;
ptrD = ptrD1; //remember this one?
delete ptrD;
delete ptrD1;
```

Output:

```
Assertion failed: isValid(), file
RNLI.CPP, line 12
Abnormal program termination
```

Clearly what has happened is that one object is deleted twice via the *Base* pointers. Previously, there was no indication that anything was wrong and the program appeared to be operating correctly. That is precisely the problem, since that can take a great deal of time and effort to track down. Here, with *RNLI* providing a level of safety, a problem is identified immediately. The diagnostic information would help to track the problem back to the offending line of code which in this instance is looking for **delete** statements.

It could be argued that aborting a program (or a controlled crash) is a little drastic and a nicer mechanism should be used instead. Fine – the bottom line is to find those buglets as quickly and as early as possible. Later, additional checks will be introduced so that *RNLI* can find the errant line(s) of code even more quickly and perhaps prior to a crash. An example could be by taking a snapshot of memory (showing the contents of *RNLI*'s collection).

Memory allocated with new

Milestone three is to differentiate between the mechanism used for memory allocation. To simplify the issue considerably, memory is allocated in one of three ways:

1) Statically, or before a program starts
   #include *"myclass.h"*
   MyClass instantiatedObj;
   int *main*() { // *whatever*
   }

2) Heap, allocated dynamically with **new**
   #include *"myclass.h"*
   int *main*() {
     MyClass aPtr* = **new** *MyClass*;
   }

3) Stack, or local variable, usually allocated for a function
   void *MyClass*::*foo*(*MyClass*& *arg*) {
     MyClass temp;
   }

There are plenty of variations of these three themes but that would only complicate the matter. Discussing memory alone could take up an entire article! However, for practical purposes *RNLI* only differentiates memory allocated with **new**, everything else is lumped together.

To do this requires overloading the operators **new** and **delete**. At times there are very good reasons (usually performance) for taking control of memory allocation, however, it is not for the faint hearted! The following implementation *is* for the faint hearted! It is very simple and assumes a vanilla setup with no other overloads of the global **new** operator.

```
class RNLI { // as before plus...
public:
  void* operator new(size_t size);
  void  operator delete(void* ptr);
private:
  static bool newAllocation;
  bool newAlloc;
  static RNLI* rnliStack;
};
```

The class declaration now includes operators **new** and **delete** to help determine from where memory is allocated (heap or elsewhere). As a rule of thumb, if the **new** operator is modified then the **delete** operator will also need to be modified. The *newAllocation* is a simple boolean flag indicating when **new** is used. The reason it is declared **static** is to enable the operator **new** to access it.

If it was declared without **static** the error message is:

- Member newAllocation cannot be used without an object

Perhaps the next 'obvious' thing to try is:

```
this->newAllocation = true;
```

However, this also fails:

- 'this' can only be used within a member function

Using *newAllocation* as a global variable is neither appropriate or necessary since using it as a static member provides the required functionality. The *newAlloc* is the local (to that *RNLI* object) version of *newAllocation* enabling the destructor to remove the object from the correct

list. The destructor simply verifies via *newAlloc* whether the object was allocated via the heap or not before destroying it.

RNLI memory allocation implementation

As usual the static variable must be initialised separately:

```
bool RNLI::newAllocation = false;
```

In line with our faint heart approach, the **new** and **delete** simply call the global version with the one exception of **new** setting the static flag to indicate that **new** was used to allocate the memory.

```
void* RNLI::operator new(size_t size) {
  newAllocation = true;
  return ::operator new(size);
}

void RNLI::operator delete(void* ptr) {
  ::operator delete(ptr);
}
```

The constructor becomes a little more complex; first, initialise RNLI's *me* with **this** from the object under construction. Next check the static flag (*newAllocation*) to add *me* to *rnliHeap* or *rnliStack*, set *newAlloc* as required and then signal the fact with a simple message. Finally set the static *newAllocation* to **false** if the object was allocated from the heap, i.e., *newAllocation* was **true**. This is necessary to ensure that subsequent objects will be added to the correct list.

The destructor is very similar except it uses *newAlloc* to decide from which list to remove the object.

```
RNLI::RNLI() : me(this) {
  if (newAllocation) {
    next = rnliHeap;
    rnliHeap = this;
    newAlloc = true;
    cout << "RNLI Heap constructor"
         << endl;
    newAllocation = false;
  } else {
    next = rnliStack;
    rnliStack = this;
    newAlloc = false;
    cout << "RNLI Stack constructor"
         << endl;
  }
}

RNLI::~RNLI() {
  assert(isValid());

  if (newAlloc) {
    rnliHeap = me->next;
    cout << "RNLI Heap destructor" <<
endl;
  } else {
    rnliStack = me->next;
    cout << "RNLI Stack destructor"
           << endl;
  }

  me = 0;
  next = 0;
}
```

Roger has tripped over an interesting bug here – can anyone see what it is? Consider the following code fragment:

```
Base* p1 = new Derived;
Base* p2 = new Derived;
delete p1;
delete p2;
```

What happens to RNLI::rnliHeap in each destructor call? – Ed.

Test the latest version

Now that the memory routines are in place it is time to look at the output again with a simple test program:

```
int main() {
  cout << "Create D on stack" << endl;
  Derived D;
  cout << "Create ptrD on heap" << endl;
  Base* ptrD = new Derived;
  delete ptrD;
  cout << "Scope rules delete stack
item"
       << endl;
  return 0;
}
```

The output:

```
Create D on stack
RNLI Stack constructor
Base constructor
Derived constructor
Create ptrD on heap
RNLI Heap constructor
Base constructor
Derived constructor
Derived destructor
Base destructor
RNLI Heap destructor
Scope rules delete stack item
Derived destructor
Base destructor
RNLI Stack destructor
```

This is better. Perhaps adding pointer location would be useful? Putting this code into the *RNLI*'s constructor and destructor would do that:

```
cout << "location: " << me << endl;
```

Summary

RNLI is coming along nicely. It now has the capability to verify objects, differentiate heap allocated objects and add or remove the debug statements using macro magic. In the process, items such as static variables, the canonical class,

operators **new** and **delete** and virtual destructors have been touched on.

That's it for part 2, the last part will extend *RNLI* to:

- Provide some heap walking capability to "see" what's in memory
- Provide some random check capability within *main*
- Output debugging information to a file

Roger Lever
rnl16616@ggr.co.uk

References

[1] Addison-Wesley, "C++ FAQ Frequently Asked Questions", Marshall Cline & Greg Lomov

[2] Addison-Wesley, "Advanced C++, Programming Styles and Idioms", James Coplien

A deeper look at copy assignment

by Uli Breymann

Note: This article was given as a talk at the ACCU meeting at Object World, Frankfurt '95, 9th Oct 1995

Introduction

Sometimes copy assignment of objects of derived classes is a nontrivial task. Based on a pattern for classes which need a special (i.e., not system generated) copy constructor, destructor, and assignment operator, some possible pitfalls are shown and solutions presented. An interesting result is that a safe virtual assignment operator in the presence of virtual base classes is first made feasible with the introduction of **dynamic_cast** into the language.

Some classes need a special copy constructor which means that it is not generated by the system. In general, these classes also need a special destructor and a special assignment operator. This article concentrates on such classes only and uses the most simple case in the examples: each class has a private **int*** as pointer to the data, the data being simply an **int** object, which has to be created in the constructor and destroyed in the destructor.

The copy constructor initialises an object with the contents of another object by allocating memory and copying, the destructor destroys the object's content, and the assignment operator needs both operations: first destroy the old contents of the object, then construct it again. In order not to write the code for destroying and constructing twice, a pattern is presented in [1], which says exactly what is meant:

```
class Thing
{
public:
  Thing(const Thing& rhs)
  {
    construct(rhs);
  }
  ~Thing()
  {
    destroy();
  }
  Thing& operator=(const Thing& rhs)
  {
    // do not assign identical objects
    if (this != &rhs)
    {
      destroy();
      construct(rhs);
    }
    return *this;
  }
private:
  void construct(const Thing& rhs);
  void destroy();
};
```

Andrew Koenig points out, that the "*example above, despite its nice structure, still conceals one nasty pitfall, which makes it impossible to apply this technique in some circumstances*." ([1], to find the pitfall was left to the reader). One of these circumstances is the use of inheritance, both single and multiple inheritance, and in fact, there is more than one possible pitfall, mainly connected with the assignment operator. The consequences of using inheritance in combination with the pattern above will be investigated in more detail. To have a working example, we complete the class *Thing*. In addition a further function *localAssign* is introduced, which combines *destroy*() and *construct*():

```
class Thing
{
public:
  Thing(int i=0)
  {
    ptrToThingData = new int;
    *ptrToThingData = i;
  }
  Thing(const Thing& rhs)
  { construct(rhs); }
  ~Thing()
  { destroy(); }
  Thing& operator=(const Thing& rhs)
  {
    if(this != &rhs)
    {
```

```
      localAssign(rhs);
    }
    return *this;
  }
private:
  void construct(const Thing& rhs)
  {
    ptrToThingData = new int;
    *ptrToThingData =
*rhs.ptrToThingData;
  }
  void destroy()
  {
    delete ptrToThingData;
  }

  void localAssign(const Thing& rhs)
  {
    destroy();
    construct(rhs);
  }
  int* ptrToThingData;
};
```

Pitfall One: The pattern has to be modified for single inheritance

Suppose there is a class *AThing* which inherits from *Thing* and has its own local dynamic data:

```
class AThing : public Thing
{
public:
  AThing(int i=0, int ia=0)
  : Thing(i)
  {
    ptrToAThingData  = new int;
    *ptrToAThingData = ia;
  }
  AThing(const AThing& rhs)
  : Thing(rhs)
  { construct(rhs); }
  ~AThing()
  { destroy(); }
  AThing& operator=(const AThing& rhs)
  {
    if(this != &rhs)
    { // modification:
      // base class part
      Thing::operator=(rhs);
      // same as before:
      // only local data
      localAssign(rhs);
    }
    return *this;
  }
private:
  void construct(const AThing& rhs)
  {
    ptrToAThingData = new int;
    *ptrToAThingData =
              *rhs.ptrToAThingData;
  }
  void destroy()
  { delete ptrToAThingData; }
  void localAssign(const AThing& rhs)
  {
    destroy();
    construct(rhs);
  }
  int *ptrToAThingData;
};
```

We see at once that the pattern is broken in the assignment operator. *localAssign*() only refers to class-local data, but clearly the *Thing*-subobject within an *AThing*-object has to be assigned to also. Obviously there is a missing feature: *Thing* is not designed for inheritance. This aspect will be discussed after looking at multiple inheritance.

Pitfall Two: The pattern has to be modified even more for multiple inheritance

Consider the following inheritance hierarchy (see Fig. 1), where there is one virtual base class *Thing* and some other classes. “Virtual” means

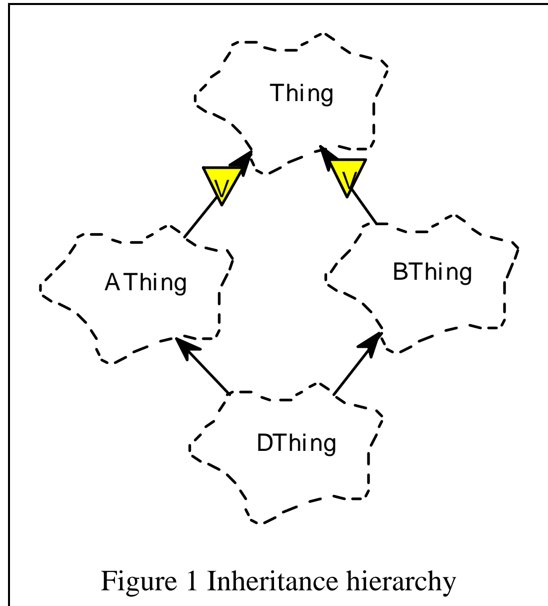

Figure 1 Inheritance hierarchy

that there is only one *Thing*-subobject in every *DThing*-object. This one subobject is common to the *AThing*-subobject and the *BThing*-subobject of a *DThing*-object.

As we do not discuss here system-generated assignment operators, it is assumed that all classes in Fig.1 have local data requiring a special copy constructor, destructor, and assignment operator. For the sake of simplicity we presume a similar structure, i.e., *BThing* has a private data member *ptrToBThingData*, *CThing* has a private data member *ptrToCThingData* and so on. Of course the constructors have to initialise the base class subobject of type *Thing*, if a concrete object of one of the classes is defined in a program. The most derived object, also called the complete object, is responsible for the virtual base class initialisation to avoid inconsistencies ([2], 12.6.2). If the complete object does not initialise the virtual base class subobject, the default constructor is called. Further initialisations (e.g., by the constructor of *BThing*) are ignored, i.e., do

not take place. As an example only class *DThing* at the bottom of the hierarchy is shown in detail:

```
// now virtual inheritance
class AThing : virtual public Thing
{ // rest as before
};

class BThing : virtual public Thing
{ // rest like AThing
};

class CThing : public AThing
{ // similar to AThing
};

class DThing
: public CThing, public BThing
{
public:
  DThing(int i =0, int ia=0, int ic=0,
         int ib=0, int id=0)
  // initialisation of base class
  // subobjects, including Thing
  : Thing(i), CThing(i, ia, ic),
    BThing(i, ib)
  {
    ptrToDThingData = new int;
    *ptrToDThingData = id;
  }
  DThing(const DThing& rhs)
  // initialisation of base class
  // subobjects, including Thing
  : Thing(rhs), CThing(rhs),
    BThing(rhs)
  { construct(rhs); }
  ~DThing()
  { destroy(); }
  DThing& operator=(const DThing& rhs)
  {
    if(this != &rhs)
    {
      CThing::operator=(rhs);
      // CThing subobject
      // second modification!
      // local BThing data
      BThing::localAssign(rhs);
      // only local data
      localAssign(rhs);
    }
    return *this;
  }
private:
  void construct(const DThing& rhs)
  {
    ptrToDThingData = new int;
    *ptrToDThingData =
              *rhs.ptrToDThingData;
  }
  void destroy()
  { delete ptrToDThingData; }
  void localAssign(const DThing& rhs)
  {
    destroy();
    construct(rhs);
  }
  int *ptrToDThingData;
};
```

What do we see? The pattern of the assignment operator is modified again! Remember the rule for initialising base class subobjects only from a complete object – there are similar reasons here. According to the *AThing* pattern above, the base class assignment operators could be called:

```
// buggy
CThing::operator=(rhs);     //  CThing
part
BThing::operator=(rhs);     //  BThing
part
```

But as we have seen, the assignment operator also copies the subobjects, including the virtual *Thing*-subobject.

That means:

```
CThing::operator=() calls
                    AThing::operator=()
AThing::operator=() calls
                    Thing::operator=()
and
BThing::operator=() calls
                    Thing::operator=()
```

so that *Thing*::**operator=()** is called twice! This is certainly not wanted and may be plainly wrong. Calling *Thing*::**operator=()** twice would be correct only if *Thing* was a non-virtual base class. For virtual base classes (here *Thing*) **operator=()** must be called at most for one base class (here *CThing*), and all other base class initialisations (here *BThing*) should use their corresponding local initialisations. Of course, *BThing*::*localAssign*() can no longer be **private**! The method has to be **protected**, which is the third modification to the pattern. Some of these aspects are discussed at length in [3]. The discussion shall not be repeated, but the resulting standard recommendations are:

- **operator=()** performs a complete object assignment.
- There should be **protected** member functions in all base classes to allow for assignment of local data.
- **operator=()** is responsible for assigning the virtual base class part.

So the operator for assigning a *DThing* could be written like

```
DThing& operator=(const DThing& rhs)
{
  if(this != &rhs)
  {
    // Thing subobject
    Thing::operator=(rhs);
    // AThing local data
    AThing::localAssign(rhs);
    // BThing local data
    BThing::localAssign(rhs);
    // CThing local data
    CThing::localAssign(rhs);
    // DThing local data
    localAssign(rhs);
  }
  return *this;
}
```

or better:

```
DThing& operator=(const DThing& rhs)
{
  if(this != &rhs)
  {
    // BThing local data
    BThing::localAssign(rhs);
    // complete CThing object
    CThing::operator=(rhs);
    // DThing local data
    localAssign(rhs);
  }
  return *this;
}
```

The idea of encapsulation is partially negated, because class *DThing* has to know a lot about its base classes and also their base classes, but there is no way to avoid it. The pattern for such cases is:

- One path in the inheritance graph from the current class to the top can be served by calling **operator=()** for one of the base classes. In this example, *CThing*::**operator=**(*rhs*) is called.
- In order to avoid multiple assignments, all other assignments have to be local assignments for the specific base classes, in our example by calling *BThing*::*localAssign*(*rhs*).

As stated above, class *Thing* lacks a certain feature: it is not designed for inheritance, and the key word for this feature is polymorphism. Until now polymorphism has been ignored in this article, and it is not addressed at all in [3], which provoked reactions from readers, discussed in [5]. Polymorphism allows us to invoke the right method for the right object at runtime. A consequence is that the behaviour of an object does not depend on the kind of access, be it via the object's name or a pointer (or reference) to the object, where the pointer (or reference) maybe of a base class type.

Pitfall Three: Is polymorphism considered?

The rationale behind polymorphism may be shown by a simple example:

```
class Base
{  // ...
  virtual Base& operator=(const Base&);
};

class Derived : public Base
{  // ...
  virtual Derived& operator=(const
Base&);
};
Derived D1, D2;
Base* firstPtrToBase = &D1;
Base* secondPtrToBase = &D2;
```

```
// as wanted, Derived::operator=() is
// called here:
*firstPtrToBase = *secondPtrToBase;
```

Without polymorphism, *Base*::**operator=()** would be called, and the semantics of the program would change if we changed the type of the pointers to *Derived******.

To achieve the desired behavior in our example classes, we have to modify class *Thing* first:

- The destructor must be **virtual** to guarantee a proper cleanup.

```
class Thing
{ // ...
  virtual ~Thing() { destroy(); }
```

- The assignment operator must be **virtual** to ensure polymorphic behavior.

```
  virtual Thing& operator=
                 (const Thing&
rhs)
  { // ... same as before
  }
  // ...
}; // end of class declaration
```

As we all know, polymorphic behavior in C++ is realised by **virtual** functions, which must have the same interface, i.e., the same name and number and kind of arguments. The return type of the functions is less restricted. It may be the same type or a pointer or a reference to the type of the class (or a base class of it), where **operator=()** is declared. Let us list the modified prototypes of the assignment operators of our example:

```
virtual  Thing&  Thing::operator=
                       (const Thing&
rhs);
virtual AThing& AThing::operator=
                       (const Thing&
rhs);
virtual BThing& BThing::operator=
                       (const Thing&
rhs);
virtual CThing& CThing::operator=
                       (const Thing&
rhs);
virtual DThing& DThing::operator=
                       (const Thing&
rhs);
```

Note that we now have the same arguments all over, instead of before **const** *AThing***&** *rhs*, **const** *BThing***&** *rhs* and so on. Now let us feed our compiler that stuff – it does not like our modifications! For example it complains within *DThing*::**operator=()**:

```
virtual DThing& DThing::operator=
                       (const Thing&
rhs)
{
  if(this != &rhs)
  {
```

```
    Thing::operator=(rhs);
    AThing::localAssign(rhs);  // Oops!
                              // type
mismatch
    // .. rest as before
  }
  return *this;
}
```

The compiler is right: *rhs* is a *Thing*, not an *AThing*, at least from a compile time point of view. What we need to know is whether the actual argument passed to **operator=()** at runtime is of type *AThing* or derived from it to maintain the normal semantics of an assignment. Possibly a C programmer would just try to cast *rhs* to the appropiate type. However, an ordinary cast from a virtual base class to a derived class is not possible, and besides, C++ has a better solution for that. It is better, because it is type-safe, and its name is **dynamic_cast**, introduced into the language in March 1993. **dynamic_cast<*T**>(*p*)** converts its operand *p* into the desired type *T** at runtime, if **p* is really a *T* or derived from *T*; otherwise the value of **dynamic_cast<*T**>(*p*)** is 0 [6]. **dynamic_cast** can also be used for references instead of pointers (see below). Instead of returning 0, **dynamic_cast** then throws an exception. Using **dynamic_cast** is depicted for class *DThing* only, but the method applies in all derived classes.

```
// argument named r instead of rhs
virtual
DThing& DThing::operator=(const Thing&
r)
{  // construct reference rhs from r
  const DThing& rhs =
             dynamic_cast<const
DThing&>(r);
  if(this != &rhs)
  {
    Thing::operator=(rhs);
    AThing::localAssign(rhs);
    // compiler is happy now!
    // .. rest as before
```

The use of **dynamic_cast** was briefly discussed in [5].

At least we now know how to avoid three pitfalls which may appear when combining copy assignment and inheritance.

Pitfall or not? Checking object identity with (this != &rhs)

The assignment operator checks the identity of objects by comparing the addresses:

```
if(this != &rhs)
{   // do something, but only if the
    // addresses differ
}
```

Is that correct in any case, especially with multiple inheritance? In the following code example we will see two different addresses on the screen:

```
DThing aDThing;
BThing &refDB = aDThing; // legal
CThing &refDC = aDThing; // legal,
                         // same object
cout << unsigned(&refDB) << endl;
cout << unsigned(&refDC) << endl;
```

What we see is actually the address of the *BThing* representation of a *DThing*. But we should not jump to conclusions or bother about memory layout; rather let's consult the ARM: *An explicit or implicit conversion from a pointer or reference to a derived class to a pointer or reference to one of its base classes must unambiguously refer to the same object representing the base class*.([2], 10.1.1)

What does that mean? Let's consider a call to

```
DThing::operator=(const Thing&);
```

using the references from above:

```
refDB = refDC;
```

By means of the virtual mechanism, *DThing***::operator=(const** *Thing***&)** is called, because the operator function is virtual and *refDB* is a reference to a *DThing* object. The right hand side, *refDC*, is converted to **const** *Thing***&** according to the rule from the ARM cited above. Within *DThing***::operator=(const** *Thing***&** *r***)**, we can be sure that the argument *r* represents exactly the object we mean. We need not discuss the **this** pointer, but what about constructing *rhs* from *r* by means of **dynamic_cast**? There is no problem: given a pointer *v* to a base class of an object, **dynamic_cast<*T*>(*v*)** returns a pointer of type *T* to that object – the identity is not lost.

If assignment operators are built correctly, then the suspected pitfall discussed is no pitfall at all. Now consider what to do without **dynamic_cast**, bearing in mind that an ordinary cast from a virtual base class to a derived class is not possible? One could think of a two-stage cast: first from *Thing** to a basic data type, e.g., **unsigned***, and then back to the desired type. Ugly, ugly! (as C-style casts mostly are) This leads to loss of information about the object identity, which cannot be restored (see results of *cout* **<< unsigned(&***refDB***)** and *cout* **<< unsigned(&***refDC***)** above).

The conclusion: in the presence of multiple inheritance and virtual base classes it is not feasi-

ble to construct a safe virtual assignment operator without **dynamic_cast**! The advantages far outweigh clumsy workarounds, despite the risk of a possible exception.

One year ago, there were opinions that use of **dynamic_cast** cannot be recommended [4, 5]. One reason for that, namely that compilers do not support **dynamic_cast**, is not true any more, as the complete example compiles and works with Borland C++ 4.5 (not with Microsoft Visual C++ 2.0. I did not test other compilers). The main reason in [4] is that Scott Myers prefers static type checking (as I normally do). He recommends "*Avoid having concrete classes inherit from concrete classes*".

Concrete class means that you can declare objects of this type, in contrast to abstract classes. Objects of abstract classes can only be subobjects of other objects, but not selfstanding objects. But sometimes there are cases, where his advice is not feasible:

- You develop a class by inheriting from a concrete library class.
- You develop a class by inheriting from an abstract class which has its own dynamic data.

The second point is more important because as was shown above, it is not the property 'concrete' or 'abstract' that leads to the use of **dynamic_cast**, but the property 'having own dynamic data' (with the consequence of needing a special copy constructor, destructor and assignment operator).

Of course, a concrete class without data should probably be an abstract class, but this is not the point here.

In a later personal communication via email, Scott Meyers clarifies his point:

> *My recommendation is to make operator= protected in base classes. That way derived class operator= functions can call their base classes' operator= functions, but general clients don't run the risk of performing partial assignments.*
>
> *In short, my advice is to make base classes abstract and to give them protected assignment operators.*

He is perfectly right, preferring non-virtual **operator=()** functions. In our *Thing*-example we don't have any abstract classes, but we also don't have the risk of a partial assignment, because according to the rule above **operator=()** performs a complete object assignment. With a virtual **operator=()** there is no chance of inadvertently doing a partial assignment. You have to write it down explicitly

```
// forced partial assignment
aDThing.CThing::operator=(aCThing);
```

but then you should know what you are doing! By the way, this statement yields no compiler error, but a runtime exception if the argument is, for example, of type *Thing*, i.e., not of type *CThing* or derived from it.

Conclusion

Sometimes copy assignment of objects of derived classes is a nontrivial task, especially if a class needs a special copy constructor, assignment operator and destructor, normally if the class makes use of pointers. Some possible pitfalls have been shown for the case of single inheritance and multiple inheritance with special respect to polymorphism and virtual base classes. Solutions were presented, including the use of run time type information (RTTI). There is no elegant solution without **dynamic_cast** since you have somehow to determine the type at runtime. The use of **dynamic_cast** shown here is therefore applicable in a similar manner for all binary functions taking a polymorphic class argument, e.g., **operator==()**.

Dr. Ulrich Breymann
breymann@alf.zfn.uni-bremen.de

References

[1] Andrew Koenig, Using constructors for assignment, C++ Report, 7(2), February 1995, p. 22.

[2] Margaret A. Ellis, Bjarne Stroustrup, The Annotated C++ Reference Manual. Addison--Wesley 1990

[3] Scott Meyers, Our friend, the assignment operator, C++ Report, 6(4), May 1994, p. 51.

[4] Scott Meyers, Code reuse, concrete classes and inheritance, C++ Report, 6(6), July-August 1994, p. 46.

[5] Scott Meyers, operator=: The readers fight back, C++ Report, 6(9), November-December 1994, p. 17.

[6] Bjarne Stroustrup, The Design and Evolution of C++. Addison-Wesley 1994, p. 308.

Change of address

by Kevlin Henney

In [1], the Harpist designed a postal address class and called for comments. Here is the class as it originally appeared:

```
class Address {
  const char* const country;
public:
  enum type { UK, US, Germany, France };
  const char* get_country() { return
country; }
  virtual void printon(ostream& =cout) =
0;
  virtual void getfrom(istream& = cin) =
0;
private:
  void operator=(const Address&);
public:
  Address(const char*);
  Address(const Address&);
  virtual ~Address() = 0;
};
```

Representing your country

The first thing to notice is that the principle query function, *get_country*, lacks a **const** qualifier. There is no actual or observable change in state so the definition should read

```
const char* get_country() const
{ return country; }
```

I am not sure what the enumeration type is doing. Or rather, I am sure that it serves no good purpose: it is not used anywhere in the interface, and would unnecessarily constrain the design of derived classes if it were. Thus we can drop it.

I will admit it's a personal preference, but I prefer attribute style naming for attribute methods. The use of get as a prefix suggests a change of state: one gets a takeaway, but one does not get a person's name from memory. It just sounds so very clunky and procedural, and quite unabstract. Just as you would prefer *size* to *get_size* and *is_in_range* to *get_whether_in_range* (or worse, *get_in_range*), in this case *country* is preferable to *get_country*. Yes, this means that in the current implementation the *country* data member needs renaming, but that should not be an issue. Remember that the public interface should be designed for the convenience of others and not as an afterthought tacked onto the implementation.

As it happens, no renaming is required: the *country* data member is surplus to requirements so we can drop it. Consider that if the intent is to partition derivation by country, the country will be the same for all objects of a given derived class. If it is the same, why are we making a copy for every object? By making the country an actual data member we have at once lost flexibility in our design and created an inefficient implementation.

The country, and its implementation, depend on the derived class. This suggests the following declaration:

```
virtual const char* country() const = 0;
```

For one of the classes suggested by the Harpist we now have the following lightweight implementation:

```
const char* UK_address::country() const
{
  return "United Kingdom";
}
```

In truth, I do not believe the Harpist's assertion that all addresses are explicitly associated with a country. Come Christmas, my relatives in Brazil will receive cards explicitly addressed to Brazil, but the addresses for my family in the UK will not contain an explicit United Kingdom. You might say that this is the basis of relative addressing.

In this case, country is not an abstract property and we can provide a default implementation in the base class:

```
const char* Address::country() const
{
  return "";
}
```

Access, your flexible friend

The declaration order in the original class seems somewhat arbitrary. A good rule of thumb is to use what I call "need to know ordering". The items which are most relevant to **public** users should go first, followed by **protected**, followed by **private**.

This raises the rather interesting issue of what to do with constructor declarations: in an abstract class they are of no use to a public user as no instances can be created. The appropriate solution to this is to declare constructors as **protected**. This is a useful recommendation [2] as it serves to emphasise the abstract nature of the class. In the absence of an abstract keyword it serves to back up the syntactically inconspicuous **= 0** suffix.

As an aside, this technique provides a useful solution to an issue in Roger Lever's article [3]. The *RNLI* debugging class is effectively a mixin base class without any polymorphic behaviour. The destructor was declared **virtual** for the sole reason that proper base classes should do this. However, you will find that most mixins need not bother. Providing a public virtual destructor is equivalent to saying "it is meaningful to delete this object through a pointer to the mixin part". Most mixins do not represent any ownership concept, and this is certainly not what you want them to advertise.

The solution is to make the destructor **protected**. The **virtual** keyword can be dropped as there is no longer an issue of publicly deleting through a pointer to base without calling derived class destructors. This will have the added benefit of making the generated code slightly lighter and, in the case of the *RNLI* class, the semantics of derived classes remain unchanged. This last point is important: the *RNLI* class is intended to be a lightweight and unobtrusive class that can be removed for production software. In the event of someone actually forgetting to make a destructor virtual this 'invisible' class should not accidentally correct it.

The *Address* class is a proper base class through which objects of derived classes may be legitimately deleted. This is a public property so we leave it declared as **virtual** near the top of the class.

Show and tell

The *printon* member is missing a **const** qualifier: I would be most surprised if writing an object to a stream changed the state of that object. If we were implementing an archiving scheme things would be different, but we're not so this is a simple query function:

```
virtual void printon
          (ostream& out = cout) const =
0;
```

Even in the archiving case, you could argue it might be reasonable to remain const and have the internal state information mutable (if the externally visible 'state' does not change) – Ed.

The naming of the I/O functions needs some attention. I am not simply referring to the absence of underscores: the names do not work meaningfully with the default arguments. Consider:

```
SomeAddress address;
address.get_from();
address.print_on();
```

Get from where? Print on what? Either you can drop the default streams – a good idea as *cin* and *cout* are highly overrated as default arguments – or rename the I/O functions. Renaming is probably not a bad idea anyway as *print* and *get* are not antonyms. There is a natural tendency to select *read* and *write*. Even though the *istream* and *ostream* classes have *read* and *write* members I am a little wary of using these identifiers: they are reserved as part of the POSIX name space and can thus be macros – I have seen them legitimately implemented as such.

I am going to look at the issue of stream naming from a different angle. What I would like to write is something like

```
SomeAddress address;
cin >> address;
cout << address;
```

As I cannot implement the stream operators for *Address* as members of *istream* and *ostream*, they must be global. The one thing they are not, however, is friend functions. This is a common mistake made by novices and experts alike. There are legitimate uses for friend classes, but friend functions often indicate a design fault. In this case the design fault is quite obvious: there is no private state to befriend and global functions are not polymorphic.

The stream operators are nothing more than syntactic sugar: the real functionality is implemented by calling member functions on the *Address* object. To some extent we are back where we started; what I have tried to emphasise here is the importance of conforming to expectation, in this case to the iostream view of the world. In this respect we can view the process of reading from or writing to a stream as a form of stream manipulation. Stream manipulators are functions, or objects that behave like functions (functors [4]), that may be called on directly on a stream or be 'streamed' on it. For instance, consider the familiar *endl* manipulator:

```
cout << endl; // write a newline and
              // flush cout
endl(cout);   // ditto
```

For *Address* we overload **operator()** to achieve the same effect:

```
ostream &operator<<(
       ostream&       out,
       const Address& address
)
```

```
{
   address(out);
   return out;
}
istream &operator>>(
       istream& in,
       Address& address
)
{
   address(in);
   return in;
}
```

The Tower of Babel

Here is the result of the discussion so far:

```
class Address
{
public: // attributes (presume to add
        // others in full design)
   virtual const char *country() const;
public: // I/O as manipulator
   virtual ostream &operator(ostream&)
                                 const =
0;
   virtual istream &operator(istream&) =
0;
public: // destruction
   virtual ~Address();
protected: // construction
   Address();
   Address(const Address&);
private: // disallowed
   Address &operator=(const Address&);
};
ostream &operator<<(ostream&,
                    const Address&);
istream &operator>>(istream&, Address&);
```

This class addresses all of the specific detailed design issues raised of the original class. Let us now consider some fundamental problems with this as a base class.

How did we intend deriving from it? The gist of the Harpist's article suggests that we should be partitioning the derived classes by country. So for our French address we can implement the country attribute as

```
const char* French_address::country()
const
{
   return "France";
}
```

What about a German address? A problem that might have been apparent to some of you earlier becomes more obvious now: where are we addressing from? If I am holding addresses for addressing from English speaking countries only, then the country attribute is "Germany". From France, it becomes "Allemagne". In effect I have created a class that unnecessarily hardwires a number of assumptions.

There are a number of routes we can consider, most of them leading to dead ends. I won't consider them here, just to say that they are examples of what Koenig calls anti-patterns [5] – where a pattern is a successful solution strategy to a generic problem.

If you are serious about modelling countries in a locale-aware manner, be sure that you are very clear about your requirements. A generic solution may prove elusive or cumbersome. Otherwise you will find that holding the country as an optionally blank uninterpreted text field is a disarmingly simple and effective solution.

The Babel Fish

Partitioning by country may prove to be less than useful in a number of cases, raising more problems than are solved. So how to divide up the address space? Take a look at a number of addresses, both in your own country and for others. See any common features? There are two fundamental ways in which addresses differ: content and layout.

To take an example of addresses that differ in content, simply consider addressing a cottage on a small island versus a department of a corporation in a shared building in a city. There is a great deal of variation here.

One way to capture it is to leave attribute handling for derived classes, and simply have the *Address* base class responsible for declaring I/O functions as pure virtuals. Assignment is not meaningful for any of the abstract classes in such a system, and should only be declared and defined in the concrete classes at the leaves of the hierarchy.

Alternatively, you can make addresses more flexible by defining all the reasonable possibilities in a fat interface. In such a scenario all attributes exist in the interface although many are optional. Additionally you may wish to define some rules governing the relationship between them. The advantage of this approach is that it requires only one class, is flexible, and is simple to map to form entry. The disadvantage is that overspecification may lead to a bloated interface, and underspecification to many changes in the future.

Layout is presentation logic. As such it is not a good candidate for modelling in the address hierarchy itself. By presentation logic I mean the possible appearance and access for a text file, a binary file, for a console, for a window, for dialog entry fields, etc. To try and put all this into a single class would be a mistake.

This is what the MODEL-VIEW-CONTROLLER (MVC) architecture (no, it's not a paradigm as it is often quoted as being) and, more generally, the OBSERVER pattern [6] address. Where there is significant variation in the way that you might change or look at a class, capture this variation outside the class with controllers and viewers. In other words, separate input and output classes. Much as you might like to believe in the symmetry of I/O, this is a forced illusion – just try convincing a laser printer to read back from the paper it just spat out, or tell your mouse to run around the desktop. *Address* concentrates on being an address and doing that one job well, with all the fuss and bother of presentation factored out into manageable and separate units.

Calling time at the bar

Another example recently outlined by the Harpist [7] can be reviewed in the light of what I have said. Given a class *Window* and a class *Text*, how do we compose a class *TextWindow*? The Harpist dismissed the idea of linear inheritance from *Window* and some kind of association with *Text*, opting instead for multiple inheritance from both *Text* and *Window*. This is unfortunate, as the first solution is in fact the appropriate one: a *TextWindow* is no more a *Text* object than an ice cream van is an ice cream, or a beer glass is a beer. The *TextWindow* is merely a way of interacting with a *Text* object – both viewing and controlling in this case – but it is not the same thing as one.

Given this separation of concerns by separation of hierarchies, you may now see a couple of ways in which you may wish to re-approach modelling countries. That, as they say, is left as an exercise for the reader.

The separation of control and view from representation responsibility is an important idea that helps shape your approach, giving you cleaner designs and allowing you to tackle other tricky problems. Another example would be handling the difference between universal (UTC) and local time: the former is the canonical representation and the model, and the latter is a view. I had a good discussion recently with Francis on more exotic calendars, and believe me there are some wild ways of counting the days out there! I will leave it to Francis to say more on the subject some time, but unless you are a historian or a religious functionary you probably don't need anything more sophisticated than a system defined in terms of UTC. Frequently asking yourself "What is the problem I am trying to solve?" should save you from the excesses of providing your clients with business time management software that handles calendars from fallen civilisations.

Conclusion

In a well defined context the address example can be useful but, as I showed before, a lack of concrete requirements will lead you nowhere useful, slowly and in circles. It seems to be a good teaching example in that it has a simple entry level, as well as more advanced depths to sink your teeth and learning into. Just beware of requirements, relevance to your system, blind alleys, and when to stop.

Kevlin Henney
kevlin@two-sdg.demon.co.uk

References

[1] The Harpist, "Addressing polymorphic types", Overload 10

[2] Taligent, Taligent's Guide to Designing Programs: Well Mannered Object-Oriented Design in C++, Addison-Wesley

[3] Roger Lever, "Simple classes for debugging in C++ – part 1", Overload 10

[4] James Coplien, Advanced C++: Programming Styles and Idioms, Addison-Wesley

[5] Andrew Koenig, "Patterns and antipatterns", Journal of Object-Oriented Programming, March-April 1995

[6] Gamma, Helm, Johnson and Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley

[7] The Harpist, "Having Multiple Personalities", Overload 8

/tmp/late/* Generating constants with templates

by Kevlin Henney

There's a lot more to templates than simple type based genericity. If you thought that containers and generic algorithms were all they were about, you may be in for a shock. Responding to Sean's

request for articles on templates, this is the first of what I hope to be a series of articles focusing on a number of the more interesting template techniques I have come across.

Sooner rather than later

I have been reading the excellent book Designing and Coding Reusable C++ by Martin Carroll and Margaret Ellis. I came across the following code and text in their discussion of macro elimination:

> *Sometimes, replacing a use of the preprocessor with an inline function requires some additional program transformations. Consider this code:*

```
#define CONTROL(c) ((c) - 64)
// ...
switch (c) {
case CONTROL('a'): // ...
case CONTROL('b'): // ...
// ...
}
```

> *Because it is illegal in C++ for a case label to be a function call, replacing the CONTROL macro with an inline function would result in illegal code. We can, however, transform the code as follows:*

```
inline char decontrol(char c)
{
  return c + 64;
}
// ...
switch (decontrol(c)) {
case 'a': // ...
case 'b': // ...
// ...
}
```

> *(Because it does the addition in decontrol at run time, this version is slightly slower than the version using the macro.)*

They go on to talk about templates and type genericity.

I'm no fan of the preprocessor: it's a dreadful tool whose only purpose in C++, in my opinion, is to include header files, compile conditionally, and be used when having to deal with poorly designed and implemented third party libraries. In their closing comment the authors unfortunately missed the possibility that the required transformation can be performed cleanly at compile time without using the preprocessor:

```
template<char raw> struct control
{
  static const char value = raw - 64;
};
```

The constant also requires a separate uninitialised definition. If your compiler does not yet support inline initialisation of static constants you will have to use an anonymous enum:

```
template<char raw> struct control
{
  enum { value = raw - 64 };
};
```

You lose the exact typing, but for the context in which we plan to use it the decay to an integer is fine:

```
switch(c)
{
case control<'a'>::value: // ...
case control<'b'>::value: // ...
// ...
}
```

We have expressed very early binding for abstracted constant calculations, and inside rather than outside the language proper. In effect this use of templates emulates the idea of templated constants. The struct is being used here purely as a scope mechanism; a vehicle for implementation. If templated namespace definitions were possible we would use these in place of the stateless struct. Although not a plain C struct, it fails our litmus test for an encapsulated object type and hence we favour struct over class.

Power rangers

Every now and then I come across the need for a relatively complex compile time calculation for something like an array size. By "relatively complex" I mean something more than a simple self explanatory arithmetic expression. For instance, flattening a fixed branch, fixed depth tree into an array. It might be a short summed series or progression, but inevitably it's me rather than the development or runtime system that performs the calculation. And me that inserts a comment to explain what I think I'm doing.

Raising two to the power of a natural number is easy: just shift it. Other numbers tend to be a little more reluctant to fit into the binary world. Naturally I would still like to make the compiler do the work for me. Here's how:

```
template<long radix, int exponent>
struct power
{
  static const long value =
    exponent == 0 ? 1
    : radix * power<radix,
                    exponent -
1>::value;
};
```

If you've not seem them before, you're probably saying "Whoa! You've got to be kidding: recursive templates?". You'd better believe it!

The Draft Standard defines a depth limit of seventeen to the recursion. This prevents potentially infinite recursion in the compiler and traps expressions that are most likely to be errors. In this case, domain errors:

```
const long illegal = power<3, -
1>::value;
                       // fails to
compile
```

In fact, the depth limit is only an implementation quantity – a minimum requirement that compilers must support. The above example isn't strictly illegal but is unlikely to be compilable on any machine with a fixed resource limit – Ed.

By choosing the widest signed representation, the compiler also has the opportunity to catch range errors in the event of integer overflow. Note that floating point types cannot be template parameters, so there are limits to your template creativity.

I have used the conditional expression to bottom out the recursion, but an alternative technique is available in the form of partial template specialisation. As we already have a simple working example, I will leave specialisations for another time.

Summary

The technique described here can be used for specifying all kinds of values as compile time constants: simplifying bit sets, summing short series, and calculating the inevitable factorial, to name but a few. In most places where C required a compile time constant C++ is more lenient, but there are still a few cases where they are needed: array sizes, enumeration constants, case labels, bit fields, and template parameters.

The more general technique that this article has illustrated is sometimes known as meta-programming. Literally, we are executing code to achieve results before runtime. It is an area I will be returning to in future articles.

I hope I have demonstrated to you that not only is the C preprocessor unnecessary for generating derived compile time constants, but that it is in fact inferior to the template solution.

Kevlin Henney
kevlin@two-sdg.demon.co.uk

editor << letters;

The only person who wrote to me this time was Andrew King of Microsoft – and he only wrote to complain that I was having an unfair dig at their compiler! Think of *Overload* when you're writing your xmas cards...

Hi Sean, thanks for running the news item on Visual C++ 4.0.

Just a couple of comments on your comments:

- page 15, article on namespaces. You commented that only Metaware supports this – but as you mentioned on Page 29 – Visual C++ 4.0 also now supports namespaces
- page 26 – your comment re: STL support in MSC++, that the customer was on a loser(!) – again, as you mentioned on Page 29 – Visual C++ 4.0 ships with the Standard Template Library and compiles it quite happily(!).

Do you have Chris Simons' email address so I can tell him he's on a winner if he upgrades his VC++ to version 4.0?

Thanks

Andrew King
andrewki@microsoft.com

So what missing language feature should I complain about now? :-)

++puzzle;

Francis sets an interesting challenge this time – with a **prize**! – so try not to underwhelm him with contributions or he'll be accusing me of redirecting his mail again.

Date with a Design

by Francis Glassborow

I am always on the look out for good programming exercises. What I am looking for is something that can be sensibly handled at many different levels of expertise and insight. The address class hierarchy that the Harpist hijacked from me in the last issue is good example.

Too often, people approach problems as if there is an ultimate best answer to which all others are approximations. I do not think that this is true. For example, suppose that you had never seen a quadratic equation before but urgently needed to solve one. You might quite reasonably use trial and error methods to achieve an answer (other possibilities include finding someone else who could solve it for you).

Now suppose that you are faced with not one, but a large number to solve. At that stage it would be worth looking for some general solution. You might eventually come up with a method such as completing the square. That would be more than adequate for a single batch, but what would happen if you found you needed to solve quadratic equations regularly or even get someone else to do so for you? At that stage you might encapsulate your algorithmic solution as a formula. Until relatively recently that would be the end of the trail. With the advent of modern technology we have two further options, write a computer program to do it, and finally produce firmware for a calculator to do it. Actually there is another generalisation which almost closes the circle, use graphical methods and a computer to produce progressively refined approximations for solving equations. Then, of course we set off on another journey.

At no stage is there an ultimate best solution (and there are other solutions that I have not even touched on that might be more appropriate in other circumstances). What we have is a problem and a set of resources for solving it. We need to find the most appropriate solution judged by some metric that seems reasonable to us.

Let's look at that address problem. If all you are doing is preparing a database of addresses of members of your local golf club, consideration of the problem of an international address is definitely over the top. On the other hand if you are preparing a world-wide mailing system for an international company you need to consider the problem of the country from which a letter is being dispatched as well as the one too which it is being sent. There is no point in beautifully addressing a letter in flowing Punjabi if you write the country in Punjabi as well. The country of dispatch may eventually determine what language the address is in, but you will be very lucky if that happens very quickly. In other words, efficient international addressing requires different parts of the address to be in different languages.

Now approach this problem from a different direction. It will do very little for the novice programmer to present them with a fully worked out hierarchy for international mailing. Even if they can understand what you are doing they will be completely befuddled by the methods you are using. The ideal problem is one that can be legitimately targeted at different levels of competence. No one wants to do something that could not conceivably be of use, yet the less experienced does not need to be defeated by design issues that are well beyond their competence.

When we present answers or just discuss the design of something such as an address type we need to provide a range of solutions so that our readers can see that there are good solutions at all levels. I think that this is something at which many of us 'experts' are very bad. We make matters worse by treating the most complicated version we can personally tackle as being the 'best' solution. I think some of us need to be willing to

present a wider range of answers without denigrating the ones that are less demanding.

I, for one, would much like to see a carefully documented, jargon free, working out of the design of an address type. Even more would I appreciate a multi-layered one that avoided being overly critical of the simpler layers. I get a little tired of people who start off ‘You don’t want to do that ... when you use that next year you will find ...’ What I want is a helping hand to the next level of programming and an appreciation that solving today’s problem adequately is a good deal better than producing bug-ridden, ill-designed solutions through over-reaching.

So now to a problem for all of you, and one that I hope will generate good, well argued solutions at all levels.

The basic problem is to design a date class – simple did I hear you say? Yes, but I want to see designs at all the following levels (together with some implementation guidelines and justification for choices):

- A simple class that will handle dates in a single format.
- A class that will handle dates in multiple numerical formats (e.g., UK, US and Japanese)
- A class that will handle non-numerical formats.
- A class that handles non-numerical formats and locale based day and month information (e.g., the French names of days and months for a French locale etc).
- Finally, a full extensible multi-calendar date system (and you’d better not assume that such a calendar will even use weeks and months – I know of some really weird systems that have been used in the past).

Each of these levels has a target type of user. If you want to think about the last one, remember that historians and archaeologists often need to try to relate dates in different systems.

I am primarily interested in the design (hierarchies and class definitions), documented so that someone with a little less skill than yourself will be able to follow it. I expect to see adequate treatment of such problems as invalid dates (when capturing date data, a date will often pass through invalid states, there should be a mechanism to handle dates that are left in an invalid state.). Don’t forget that you also need to handle incomplete dates. ‘How’ is your problem.

I hope that there will be many contributions (if appropriate, some might finish up in *CVu*). Send them direct to me. I will collate them all and produce one or more articles based on them. I will arbitrarily award the one I like best a copy of the Anniversary Edition of Frederick Brooks’ “The Mythical Man Month.” In this context, “arbitrary” means that I decide and the criteria are clarity and appropriateness to the level you have chosen to tackle. If you think that there is some other level that I have not specified, you are free to specify it, give an example of a target user of the specification and then design the solution. It will be harder to produce a good solution to the more complicated levels because I will expect adequate documentation of all aspects so that most readers of *Overload* will get something from your submission even if ultimately it is using methods that are far beyond them (or even me, if the truth be known).

The deadline is January 6th (it will allow the fully employed to spend Christmas working on it). Some earlier submissions would help me get a preliminary column into the next issue of *Overload*.

Francis Glassborow
francis@robinton.demon.co.uk

The prize is certainly worth winning and the puzzle provides opportunities at all levels so get writing! – Ed.

Books and Journals

I’m still looking for a reviewer for “Foundations of Visual C++ programming for Windows 95” – if you have Windows 95, VC++2.0 (or later) and a CD-ROM drive, please drop me a line.

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

Scientific and Engineering C++
reviewed by Sean A. Corfield

Title:	Scientific and Engineering C++ – An Introduction with Advanced Techniques and Examples
Authors:	Barton, Nackman
Publisher:	Addison-Wesley
ISBN:	0-201-53393-6
Price:	£28.95
Format:	hardback, 670 pages

Barton and Nackman clearly love C++ – let me quote from the preface: "We think you should try C++, and we wrote this book to help you get started." If you've read the book, you'll share my amusement over this quote, but the question is: "does this book live up to its title?"

An Introduction

Despite my initial scepticism, the book provides a good grounding for non-C, or at least FORTRAN, programmers by showing the C++ equivalent for common FORTRAN constructs, explaining references by analogy to FORTRAN's pass by reference and explaining pointers with clear diagrams. Alongside this, Barton & Nackman give useful hints and tips for C++ that avoid some of the major pitfalls. The brief chapter purporting to explain C++ to non-FORTRAN programmers is less successful in my view because it ignores the "bad habits" that C programmers often carry over to C.

Classes are introduced gently using a fairly standard example (*Point* and *Line*) but their approach is to focus on design and use rather than implementation which I hope makes it easier for non-C++ programmers. By page 100 we have templates and a few pages later, exception handling. Both of these are introduced with similar focus on design and use. This focus on designing elegant, extensible solutions to a range of well thought out problems allows the authors to introduce interface classes (ABCs), relationship classes, multiple inheritance and so on in a natural progression, allowing even relatively new programmers to follow the discussion.

The section on *Object Lifetime and Memory Management* is particularly well thought out, constructing trace classes that are used to show, in detail, how objects are born and when they die, even in the presence of exception handling.

At the end of the day, however, Barton & Nackman shouldn't be your only introductory text and they agree, pointing to several other "learn C++" books.

Advanced Techniques

The middle section of the book concentrates on identifying commonality, encapsulation and making the most of OOP in C++. A straightforward, focused example is presented and repeatedly refined as more sophisticated commonality is identified and isolated using various inheritance patterns (**public** and **private** inheritance, single and multiple base classes, virtual inheritance and so on). Templates are featured heavily in ways that may not be familiar to many C++ programmers, e.g., to provide function structure commonality through a template base class.

Although their intense analysis of commonality overwhelms their examples to some extent, their techniques should be useful in real world projects if applied with care. They do note that identifying commonality for its own sake is not always productive.

Examples

The first serious example comes after the introductory chapters and concerns a "mesh" as used in the numerical solution to partial differential equations – you don't need to understand the maths because the example focuses on design and respresentation. The authors develop an "obvious" solution and then provide critiques and refinements that improve the robustness to change in a very convincing manner. They repeat this for electrical test equipment – used extensively throughout the commonality chapters – and later for increasingly sophisticated array implementations.

Quite often the authors omit initialisations, constructors and so on but they explain exactly what the compiler does in these cases and justify why they choose to rely on the generated defaults. I was somewhat nervous of this as a general technique but there are some places where it is actu-

ally safer to follow their lead, e.g., multiple inheritance with virtual bases.

Scientific and Engineering C++

The third section of the book covers various scientific application areas (LAPACK, data modeling, dimensional analysis, groups and rings, 2-d and 3-d arrays and projections). Some of this material has appeared in their regular column in the *C++ Report* and although it contains many interesting techniques, parts of it are of much more specialised appeal than the rest of the book.

In the light of STL and the fervour that has generated, it is interesting to note that Barton and Nackman introduce iterators as part of their exposition of arrays with similar justifications – to produce algorithms that work independently of the container being operated on.

For me, the highlight of this section of the book is the chapter on function objects which quickly builds up classes for performing symbolic algebra – for simplicity and power.

Summary

For FORTRAN programmers, a good starting point but otherwise not ideal for learning C++ although since their intent is to take you beyond that level fairly quickly, I can't really criticise them for it.

What everyone will read this book for is the "Advanced Techniques" that Barton and Nackman present for expressing relationships and commonality in various forms. The authors' enthusiasm draws the reader in but at times that enthusiasm runs away with them and they make mental leaps that can take a few readings to catch up with – I sometimes needed to put the book down simply to take a rest from the flow of ideas – but overall I found their style produced a readable but highly technical book.

All the code fragments are available by anonymous FTP and the authors provide an email address (via Addison-Wesley).

If you don't already own this book, buy it now!

Sean A. Corfield
sean@corf.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.

The following news item is taken from Take-Five's newsletter – SNiFF+ is an open-architecture, integrated development environment. Perhaps we'll see a Java parser as part of SNiFF+ soon?

SNiFF+2.1

We'll be adding significant functionality in this release. Look for:

- Multi-Programming-Language-Support
- Symbol table API: enables programmers to create their own applications through access to underlying symbol information
- DDE and dbxtra debugger integration: further expands SNiFF+'s openness in including a range of the most popular debuggers
- 30-50% faster project loading times
- SNiFF+ on SCO Unix, Novell UnixWare and Linux as product available

Family of programming-language parsers

For the first time, projects that include different languages can be edited and managed in ONE integrated development environment. All object-oriented languages that have concepts similar to C/C++ or that are extensions of C/C++, e.g., IDL, 4GL (TCL, PERL, Python, etc.), as well as common procedural languages like FORTRAN or COBOL, can be integrated thanks to SNiFF+'s language-independency. No other development environment offers this feature.

There are many ways of taking advantage of SNiFF+2.1's language-independency. For example, companies can integrate source code from any given language into SNiFF+2.1. On the other hand, software systems which are already available can be re-implemented and therefore integrated into an object-oriented C++ development environment. All in all, language-independency will make SNiFF+'s C/C++ de-

velopment environment more universally applicable!

In order to also support proprietary languages, SNiFF+2.1 can be used in conjunction with an Open Parser API, thereby allowing SNiFF+ users to write their own parsers. The Open Parser API will be available for all SNiFF+ versions including and subsequent to SNiFF+2.1.

Besides offering customers the possibility of writing their own parsers for proprietary languages, TakeFive Software will also make available a family of programming-language solutions.

TakeFive Software GmbH, Salzburg, Austria, announced at the GUUG95 trade show in Wiesbaden, Germany, that SNiFF+2.1 can be extended with an IDL-Parser through a cooperation with Interactive Objects Software, Elzach, Germany.

Interactive Objects Software has been selected as TakeFive's partner for developing the IDL-Parser. Interactive Objects has an in-depth knowledge of CORBA development due to its distribution of and consultancy work for CORBA products. A further aspect of SNiFF+2.1's language-independency is the ability of having multiple parsers running simultaneously and in parallel with each other.

Europe: info@takefive.co.at, +43 662 457 915
USA: info@takefive.com, +1 408 777 1440
WWW: http://www.takefive.com

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, Carthorse Lane
Woking, Surrey, GU21 4XS
john@wash.demon.co.uk

Subscriptions

Dr Pippa Hennessy
c/o 11 Foxhill Road
Reading, Berks, RG1 5QS
pippa@octopull.demon.co.uk

Distribution

Mark Radford
mark@twonine.demon.co.uk

Copyrights and Trademarks

Some articles and other contributions use terms which are either registered trademarks or claimed as such. The use of such terms is intended neither to support nor disparage any trademark claim. On request, we will withdraw all references to a specific trademark and its owner.

By default the copyright of all material published by ACCU is the exclusive property of ACCU. An author of an article or column (not a letter or review of software or book) may explicitly offer single (first serial) publication rights and thereby retain all other rights. Except for licences granted to (1) Corporate Members to copy solely for internal distribution (2) members to copy source code for use on their own computers, no material can be copied from *Overload* without written permission of the copyright holder.

Copy deadline

All articles intended for inclusion in *Overload 12* (February) must be submitted to the editor by January 8th.

FULL PAGE ADVERT GOES HERE!