ISSN 1354-3172

# Overload

*Journal of the ACCU C++ Special Interest Group*

*Issue 26*

*June 1998*

# Contents

# Editorial

## Broadening

Over the past six months I've been encouraging you authors to present material which goes beyond C++ as a language. C++ is the language I use to codify my designs, but what's the language I use to express my design, and how do I translate that into code?

All mature development processes include design and documentation phases, and being able to communicate effectively is a key engineering skill. Richard Blundell has been producing an excellent tutorial series on the UML, which embodies many techniques for exploing, problem spaces and communicating solutions.

Along side this exploration of a modelling language we've had a number of articles which document the development experience of applying C++ in various application domains. The notion of 'Software Patterns' is a formalised way of recording some aspects of this software experience. This issue Francis begins a series of articles discussing the patterns presented in the Gamma, et. al. 'Design Patterns' book. There is a great scope of work that could be explored here. How might each pattern be implemented, improved upon, or combined? Are patterns even a worthwhile thing?

*Overload* is starting to cover various aspects of object oriented technology, but from C++'s own vantage point. There are complementary or competing technologies that we should consider. For example, issues that have occupied much of my time over the past few years are componentisation and distributed object technology.

The software development environment appears to be becoming more hetrogenious then homogenious. Enterprise software systems are constructed with a variety of software tools, languages, and even operating systems. Components attempt to define the common binary gulf between packages built in different languages. An article exploring COM/DCOM, CORBA, and perhaps even Java/RMI would be of great interest to myself, and perhaps others.

*John Merrells*
*merrells@netscape.com*

## Sean's Show

I seem to have been out of touch with Overload for a long time. What's been happening in Sean's world? Enough to fill a book. Maybe enough to fill a few *Overload* articles...

I'm still "Book Review Coordinator" and despite several attempts during my time as editor, I never had much success in rounding up ACCU members who were keen to undertake in-depth reviews of C++ and OO-related books. However, the publishers are still sending them to me so I'm appealing for reviewers - I have about half a dozen reviews in progress but I can't review everything.

So, please, potential reviewers: contact me and I'll send a list of books available for review. In particular, I have several Windows 95 & NT specific books that I do not have the expertise to review so I would like such experts to volunteer. Remember that I need in-depth reviews since many of these books have already been reviewed in brief in CVu.

Finally, after all my esoteric articles on the obscure corners of C++, I'm starting a series of articles on real-world stuff: the last year has seen me wrestle with a C++ application framework for building e-commerce web sites. I intend to write about my traumas and triumphs. I've also had cause to use ObjectSpace's standard library in anger, as well as their web toolkit. I intend to write about that. And recently, my client has begun to use Java for some aspects of their web-related work. If Overload readers are interested, I will write about that too.

*Sean A Corfield*
*sean.corfield@issolutions.co.uk*

## Copy Deadline

All articles intended for publication in *Overload 27* should be submitted to the editor by July 1st, and for *Overload 28* by September 1st.

# Software Development in C++

## Make a date with C++, A Touch of Class
## By Kevlin Henney

*The purpose of abstraction in programming is to separate behaviour from implementation.*

*Barbara Liskov*

*In this connection it might be worthwhile to point out that the purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.*

*Edsger W Dijkstra*

Previous articles have concentrated on the "better and safer C" aspects of C++. It is perhaps time to take it to the next level and start focusing on what the language does to support better program construction as a whole. As the quotes suggest, abstraction is the essence of good programming, and the more a tool does to support that, the cleaner the concepts in our program become. As it turns out, the date type we have been looking at offers an excellent opportunity for abstraction.

### Open to abuse

The current representation for dates we have chosen is

```
struct date
{
  int day, month, year;
};
```

On the face of it this seems OK as it corresponds directly to the way that most people think about dates (give or take a little field ordering). However, there are some fundamental problems with this: the first is that as far as the compiler is concerned this defines an aggregate type that is composed of three `int`s. And that's it. Your understanding of the concept represented and the relationship between the fields remains undocumented in your head, and the compiler will treat this as an arbitrary structure of `int`s. OK, so perhaps you add some comments, but this has surprisingly little effect on the compiler; the type is still open to malicious and accidental misuse:

```
// completely invalid values
date nonsense;
nonsense.day   = 32;
nonsense.month = -42;

// 2 digit instead of 4 digit year
date y2k;
y2k.year = 98;

// 1900 not a leap year
date obiwan;
obiwan.day   = 29;
obiwan.month = 2;
obiwan.year  = 1900;
```

### Presentation versus representation

Another aspect that we have to consider is that *DD/MM/CCYY* is perhaps not the best all round representation for dates. An alternative approach is to use the number of days since a fixed point, typically referred to as an epoch. The two most popular epoch based representations are days since 1st January 1900 and the Julian day (number of days since 1st January 4713 BC, also known as the *star date* [1]), e.g.

```
struct date
{
  long day_no;
};
```

Consider how you might go about comparing two dates. For the *DD/MM/CCYY* version you might first try some fairly involved logic:

```
bool less_than(date lhs, date rhs)
{
  return
    lhs.year < rhs.year ||
    (lhs.year == rhs.year &&
    (lhs.month < rhs.month ||
    (lhs.month == rhs.month &&
    lhs.day < rhs.day)));
```

[1] From its use in astronomy rather than anything to do with Star Trek.

```
}
```

An alternative approach is to consider the date as a simple magnitude in a discontinuous range, e.g. 21st February 1998 can be easily translated to the value `19980221`, which is numerically greater than `19970214`, or 14th February 1997:

```
bool less_than(date lhs, date rhs)
{
  return lhs.year * 10000L +
    lhs.month * 100 + lhs.day <
    rhs.year * 10000L +
    rhs.month * 100 + rhs.day;
}
```

However, even this does not compare with the simplicity and efficiency of the day number approach:

```
bool less_than(date lhs, date rhs)
{
  return lhs.day_no < rhs.day_no;
}
```

The same issue applies to date arithmetic, e.g. adding 30 days to a given date or subtracting two dates to find the day difference. In each of these cases the day number is the simpler and more efficient representation, although *DD/MM/CCYY* is more familiar in its presentation. On the other hand, if you wish to print out a day number based representation in a more familiar format, e.g. *DD/MM/CCYY*, there is a lot more work involved than printing out the field based structure!

In truth we have two views of the calendar system that we use: one is the cycle of days within months within years, and the other is the repeating short cycle of days within weeks. The interesting – and problematic – thing is that they are in no way in synch with each other. For the day number approach there is no problem in converting to the day in week presentation: given the day in the week that the system started counting from (e.g. 1st January 1900 was a Monday) a simple piece of modulo 7 arithmetic results in the day of the week. For a field based representation you need the slightly more involved formula known as Zeller's Congruence:

```
day day_in_week(date when)
{
  int d = when.day,
    m = when.month, y = when.year;

  if(m == 1 || m == 2)
  {
    d -= is_leap_year(y) ? 2 : 1;
    m += 12;
  }

  int z = (1 + d + (m * 2) +
    (3 * (m + 1) / 5) + y + y / 4
    - y / 100 + y / 400) % 7;
  return day(z);
}
```

Where `is_leap_year` is a function and `day` is an `enum` type defined in previous articles (*Overload* 19 and 20).

Clearly, each approach has pros and cons depending on the type of application. However, the open `struct` does not provide adequate protection from any changes of design decision: all uses of a `struct` are coupled to its internal representation rather than the concept it represents – i.e. there is a global dependency on *how* it is implemented rather than on *what* it is implementing. The current design does not help us separate the issues of presentation (the concern of the programmer as the user of a type) from representation (the concern of the programmer as the provider of a type).

## Firewalls

One approach to solving this issue is to provide a type that accommodates both representations. Such a type was defined in a previous article (*Overload* 20) to illustrate the C++ anonymous union. However, in practical terms this is wholly unsatisfactory: all functions that use the type must now implement `switch` code to handle both representations, increasing the type's complexity significantly. It is also unlikely that the user of the type wishes to be presented with such complexity, or indeed need both representations; so long as the basic operations available perform their expected function that will satisfy most needs.

Given that we know what we want from a type in terms of its behaviour, i.e. the operations on it that we consider primitive and common, we would like our program to deal with the type in terms of functions and have some protection from representation issues. In other words, define a type by the operations on it rather than by its representation: an *abstract data type* (ADT) is the name traditionally given to such a type. This drives the development from the type user's perspective (i.e. the client) rather than the supplier's (i.e. the type provider). Primitive and common operations for a date type might include some of the following: initialisation, comparison, arithmetic, conversion to and from a stringified form, etc. Therefore, structure the program by placing the type definition in a header along with these operations.

However, the semantics of an object defined by an ADT are not guaranteed if its concrete representation is in anyway tampered with, so how can we ensure the integrity of the representation is not compromised? Well, here's one approach:

```
// please do not access data members
directly struct date
{
  int day, month, year;
};
```

Whilst certainly polite, I will let you judge for yourself how successful a strategy a simple comment is! What is needed is a more effective way of throwing a firewall around the concept's representation within the language, remembering that a firewall is a secure mechanism for limiting both accidental and malicious damage. The concept of an opaque type (a term borrowed originally from Modula-2) is a forward declared `struct` in a header whose full definition is only given in the same source file as the definition of all its associated functions:

```
// header file
struct date;

bool set(date *,int dd,int mm, int ccyy);
bool in_leap_year(const date *);
int day_in_month(const date *);
int month(const date *);
int year(const date *);
...
```

```
// source file
struct date
{
  int day, month, year;
};
...
```

It is now impossible for the client to mess with a `date` object or depend on its representation. There are a couple of stylistic and pragmatic issues to note in the code above:

All of the operations operate on an object of the type of interest.

Some of the queries may correspond to simply returning a value. Although trivial, they are important and primitive operations that ensure the type is both usable and that its representation is inaccessible. For example:

```
int day_in_month(const date *when)
{
  return when->day;
}
```

Operations that correspond to queries operate on a `const date *` rather than a `date *`. The importance of `const` correctness was covered in the previous article (*Overload* 22).

As the `struct` is not fully defined, it is not possible to define variables of that type. This means that all of the operations must operate on pointers – the `sizeof` a pointer to a `struct` is always known, even if the `sizeof` the type is not.

Another consequence of the type opaqueness is that objects must be allocated dynamically. As their size is not known outside the source file defining `date`, this means that part of the primitive set of operations must include a function that allocates and initialises a date object, and one that deallocates it. A refinement to this is the ability to make duplicate copies of an object.

Increased use of dynamic memory can have an impact on the efficiency of a program.

The complete hiding of the `struct` definition also means that inline functions – if needed – requiring access to the data members cannot be defined in the header, and therefore cannot be used.

The use of dynamic memory makes the type slightly harder to use. It is no longer simply a matter of declaring an `auto` variable, an object must be allocated at initialisation and cleaned up explicitly once finished with.

Some of these constraints mean that we cannot use the opaque type approach in all circumstances, although it does an excellent job of support information hiding and is not used nearly enough in C programs.

## Firewalls with class

It is likely that for a type as fundamental as a date we would not want some of the awkwardness associated with opaque types, but we still wish to erect a firewall. C++ provides use with the class construct. This time I will restrict myself to presenting the class from the class user's point of view; in future articles we will concentrate on the implementor's side:

```
class date
{
public:
  bool set(int dd, int mm, int ccyy);
  bool in_leap_year() const;
  int day_in_month() const;
  int month() const;
  int year() const;
  ...
private:
  int dd, mm, ccyy;
};
```

Here the operations that define the behaviour of the type are placed within the type definition. Placing functions within a type looks a bit weird at first if you've come from a straight C background, but using them is no different to using any other member, i.e. `.` and `->` notation still applies. You just have to think that the operation is operating on the object it is qualified from:

```
date when;
when.set(21, 2, 1998);

cout << when.day_in_month() << '/'
     << when.month() << '/'
     << when.year();
if(!when.in_leap_year())
{
 cout << "It's not a leap year!" << endl;
}
```

From a linguistic point of view we can say that the class defines the vocabulary for the type. The object that is being acted on is the receiver of a request for an operation, and sometimes you may see the term *distinguished receiver* being used to refer to this concept. The idea that object-orientation introduces is that operations are applied to a significant object, rather than the functional view where functions are the primary building block and objects are passed into them. This sort of inversion is something that you will see often when comparing object-orientation to structured approaches.

## D'you know what I mean

Again, from the linguistic point of view we might regard each whole invocation as a statement, the object being acted on as the subject of that sentence, the operation as the verb, and any arguments as the rest of the predicate (other objects, adverbs, etc.). A key benefit of a strong type system is the ability to express what you mean in precise enough terms that the compiler can verify at least some of the basic usage. For instance, attempting to call an operation on an object that is not defined by its class will result in a compile time error.

If we return to the class definition for date there are a couple of unanswered questions: `public` and `private`; the use of `const` after a function declaration. Both of these issues enforce concepts that we introduced earlier:

`public` and `private` are access specifiers, and the compiler will check your use of members against them. The `public` section forms the interface to the class and is accessible and usable by all. The `private` section is where you place your representation; any attempt to access this

from outside the class will result in a compile time error, e.g. `when.dd`.

The `const` qualifier is placed after every member function that represents a pure query; i.e. does not modify the state of the object it is called on. This indicates to both the human reader and the compiler that the object remains unchanged and is a query – a compilable comment, if you will. Looking back at the opaque type introduced earlier you will see that these correspond to the members that were operating on pointers to `const` objects. As stressed in the previous article, appropriate use of `const` is an essential part of C++ programming; it is a specification tool that should not be ignored: say what you mean.

## Come together

So finally, to *encapsulation*. It's an essential buzzword and structuring principle behind class design. The separation of interface from implementation, and the hiding of that implementation, is traditionally known as information hiding: this we can emulate with opaque types as well as more directly using `public` and `private`. The placing together of function and data in the same unit is the other aspect of encapsulation. Literally, encapsulate means "to put in a capsule", and you can see that this is effectively what we have done. The capsule in this case is a type that may now be treated as a sealed component. We are free to change the representation at will, with only a recompilation required. There will be no impact on the user's written code as they have no way of expressing a dependency on the representation of the class.

Given many of the apparent similarities to `struct`, how do `struct` and `class` differ? In truth, we could have written the `date` class above as a `struct`: everything you can do with `class` you can do with `struct`, and vice-versa; there is only one minor difference between them. So why use one rather than the other? By convention `struct` is used for traditional data structures (plain ol' data types, or PODs) whereas `class` is used for encapsulated data types, better known in OO circles simply as classes.

The minor language difference between `struct` and `class` is in default access: by default everything in a `struct` is `public` (hopefully, after a moment's reflection, this should not surprise you); in a `class` everything defaults to `private`. When using `class` it is normal to use *need to know* ordering, i.e. in the order of most interest to the reader (`public` first and then `private`), and therefore the default access is not relied upon. As a point, access specifiers can be used in any order and repeated; some conventions for using this will be presented in a future article. Note that there is nothing but the scorn of your peers to prevent you declaring data members as `public` in a class! C++ is a language that fully supports object-oriented principles, but it does not require you to use them.

## Summary

Design is an essential element of software engineering. A higher level of abstraction tends to be the driving force behind most considered design decisions.

Separation of interface and implementation is a fundamental software engineering principle. Abstract data types offer a simple way of reasoning about a program in this way. The simplest way of constructing ADTs in C is through the use of opaque types.

C++ offers extensions to C that support encapsulation more directly. Encapsulation is a cornerstone of object-oriented techniques, and is a specific application of information hiding techniques.

The essential difference between `class` and `struct` at the language level is their default access. Their most significant difference is in their convention for use. `class` is used for defining encapsulated data types with `public` member functions – `const` where necessary – and `private` data.

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

# Dynamic Function Calling Using Operator Overloading

By Richard Blundel

## Introduction

Have you ever wanted to write a function that can take different parameters at different times, depending on the circumstances? Or maybe a function that can operate on parameters generally, and so can accept any number of parameters in any combination? Or even a function that can produce a range of different return types depending upon how or when it is called? (C++ won't let you overload functions on their return types.) Some of these suggestions are certainly not standard C++ style, but sometimes you need to achieve such effects, and the extensibility of C++ can help you to do so. I'll start by mentioning a few examples of circumstances where you might want to use such facilities, and then describe a possible framework that allows you to do any of the above in a fairly neat and familiar syntax using constructor- and operator- overloading.

## Printf and Dump functions

I dare say that almost all of you will have at some point used functions that take a variable number of parameters. The C library functions printf, sscanf, etc., are common examples, but others include many in-house and third-party debugging functions (Dump, assert, etc.). These functions typically accept a variable number of parameters of unknown (at compile-time) type by using the <varargs.h> part of the C library. These library functions allow you to pull values of known type off the parameter list one at a time.

Unfortunately, it is not possible for the callee to query the type of each parameter using these functions. This has led to two popular methods of operation. The simplest case is to assume all the parameters are of the same type, and the last value holds some form of sentinel value. The callee then simply reads parameters of the known type until it reads the sentinel, and then knows it has got them all. The more common, if slightly more complicated and powerful, method is to pass in, often as the first parameter, a value of known type, such as a string, which encodes the number and type of all the following parameters. The callee then parses this string to determine what parameter types to request. The printf and sscanf functions operate in this way.

It can often be a bit of a pain that these functions require you to specify this string. It is an extra burden on the client to set this information up, and can also be a source of many dangerous errors, because these calls are not type safe. If a user passes in a set of parameters that does not exactly match the specification string, the behaviour of the program is undefined (and it usually crashes by corrupting the stack!).

Alternative methods exist, such as using the C++-style insertion and extraction operators with a stream or some other object that needs to receive the parameters. These operators allow type-safe parameter passing, but do not always meet other requirements. An Assert function may require a Boolean parameter first to say whether or not to assert and display the subsequent parameters. Similarly, insertion and extraction operators are usually stateless, making it hard for the callee to operate on a number of its parameters at the same time.

Before moving on to a proposed solution to this problem, let's look at a couple of other cases that throw up similar problems of variable parameter lists.

## Thunking

Thunking is a technique used to communicate between processes in different address spaces. On Windows platforms it is used to allow 16-bit and 32-bit processes to work together, usually for the support of legacy modules that cannot be recompiled in 32-bits, such as third-party libraries.

The most common way to perform this is to funnel all calls from the 32-bit side through a

single thunking function. Suppose we have a 32-bit application that needs to call a number of functions in a 16-bit DLL. Typically, you would write 32-bit proxy functions for each DLL function that you want to call. These functions first package up the required function parameters. They then pass this package to a single thunking function, along with an ordinal number to denote which of the DLL functions is to be called. The thunking function *marshals* the parameters to the 16-bit side of the thunk using operating system calls or messages. It also converts the parameters if necessary, narrowing integers and allocating memory for strings, buffers, etc. The 16-bit side of the thunking function then unpacks the parameters, and typically has a huge switch statement to direct the parameters to the correct 16-bit function. Return values need to be packaged up in a similar way for the return journey, although this is made easier if the functions only return simple integer types.

Often such thunking mechanisms include an awful lot of code for proxy functions that all do much the same thing, and code to package parameters up and unpack them, etc. All this code is often hard-coded, hard to check, and difficult to maintain if DLL interfaces change, or if additional DLL functions are later required.

## OLE/COM and IDispatch

A related overhead is sometimes involved in calling COM dispatch interfaces from C++. A dispatch interface, often referred to as IDispatch, is a COM interface containing helper methods that allow other methods within the interface to be dynamically called at run time (using the 'Invoke' method). There are many advantages of determining and resolving method calls at run time, and languages like Visual Basic use this form of calling (almost) exclusively, but it has its disadvantages. C++ is a strongly-typed language, and it does not allow functions to be called with parameter lists that are not known at compile time (with the exception of the C-style varargs method discussed earlier). As a result, all the parameters need to be packaged up into an array by the user before being passed to the Invoke method, along with a similar array of type information values – again, an error-prone procedure.

## Encapsulating type information

Some of the problems above can be circumvented if only the parameters of a function could have dynamic types. Why does a function parameter need to be an int when it could be a user-defined type (UDT) that contains an int, and that knows it contains an int, but that can hold other types as well? If we had such a class we could truly have our parameters behaving like dynamically-typed objects. The outline of just such a class is shown in listing 1. In the example it can cope only with parameters of type int, double, char * and const char * (so it can be const-safe as well), but other simple types[2] can be added easily.

Using this class, we can define functions that take a parameter of this type, or a reference to one, and then any of the types VarArg knows about can be passed to the function. The called function can query the type of the parameter using the type() method, and can extract the contained value using one of the getX() accessor methods. If you are into exceptions, you could put checks into these accessor methods and throw an exception if a query method is used on an object of a different type. Other enhancements would include some limited conversion methods so that, for example, getDouble() could return the double-value equivalent for a VarArg object containing an int.

Note that the non-explicit constructors provide free conversion from the contained types. Conversion operators (ctors with single parameters) are usually frowned upon, but in this case they are helpful – a VarArg object is supposed to be able to represent each

[2] Unfortunately, UDTs with ctors or dtors cannot be held in a union – a property assumed by this implementation – so additional steps are required to hold types such as strings and other classes ( see the section on Safety Issues for more details).

of its contained types, and so implicit conversions are what we want here.

## Handling variable numbers of parameters

So, this is a start. We now need to be able to pass an unknown (at compile-time) number of parameters to a function. To achieve this, I turned to the standard library. What I wanted was a self-describing parameter array with as natural a syntax as possible. The result is shown in listing 2. The ParmArray class acts as an array of VarArg objects.

```
class VarArg
{
public:
  class invalidParameterAccessException;
  enum ArgType
  {INVALID=0,INT,DOUBLE,PCHAR,CPCHAR};

  VarArg()
    {memset(this, 0, sizeof(VarArg));}
  VarArg(int val)
    : m_type(INT), m_int(val) {}
  VarArg(double val)
    : m_type(DOUBLE), m_double(val) {}
  VarArg(char *val)
    : m_type(PCHAR), m_pchar(val) {}
  VarArg(const char *val)
    : m_type(CPCHAR), m_cpchar(val) {}

  ArgType type() const
    {return m_type;}

  int getInt() const
    {return m_int;}
  double getDouble() const
    {return m_double;}
  char *getPChar() const
    {return m_pchar;}
  const char *getCPChar() const
    {return m_cpchar;}

private:
  ArgType m_type;
  Union
  {
    int m_int;
    double m_double;
    char *m_pchar;
    const char *m_cpchar;
  };
};
```

**Listing 1 – The VarArg class that encapsulates type information**

```
#include <vector>
class ParmArray
{
public:
  class badParameterArrayIndexException;
  // set up method -
  // used by sender of object
 ParmArray &operator,(const VarArg &parm)
    {m_array.push_back(parm);
    return *this;}

  // query methods
  // (used by receiver of object)
  int size() const
    {return m_array.size();}
  void clear()
    {m_array.clear();}

  const VarArg &operator[](int index)
const
  {return
   (index >=0 && index <= m_array.size())
     ? m_array[index]
     : s_error;}

private:
  std::vector<VarArg>  m_array;
  static const VarArg  s_error;
};

// in implementation file...
// static initialisation
const VarArg ParmArray::s_error;
```

**Listing 2 – The ParmArray class that encapsulates a list of parameters of a range of types (methods inlined for brevity)**

This class contains a vector of VarArgs, and has methods to build and query this vector. Note that I used composition rather than public inheritance here because I wanted to limit clients' access to the underlying array (I could have used private inheritance instead). The array can be emptied for reuse, its size can be queried, and parameters and their types can be retrieved from it by index. Note here that I have used a static error parameter if the index is out of range. Some may prefer to throw an exception if you are using them elsewhere in your project.

Parameters can be added to the array using the overloaded comma operator. The reason that I used the comma operator to append parameters to the array will become clear in the examples that follow. I wanted to try to make the parameter array behave as closely to a normal parameter list as possible. I was thwarted somewhat by the standard operator precedence, but apart from an extra set of

parentheses that are sometimes required, I pretty much got there.[3]

## Examples – Dump-type functions

OK, on to some example of how this can be used. Let's start with the Dump method, mentioned at the beginning, to output debugging information to the screen, a log, etc. With the example Dump function shown in listing 3, we can now write calls as shown below:

```
Dump((ParmArray(), "Answer = ", 42, "\n\n"));
```

Note that unfortunately we need the additional set of parentheses. I believe this is because if they are missing, the parentheses for the Dump function bind more strongly to the ParmArray object than do the commas, and the compiler complains that instead of one ParmArray parameter the function has been called with additional parameters tacked on the end.

The general syntax, therefore, is to use an additional set of parentheses, and prepend your parameter list with an temporary object of type ParmArray(), which I have done here using a temporary object. If you want, you can declare an object of type ParmArray, and reuse it for several function calls, calling the clear() method in between calls. Note that there are no messy type arrays or encoded string of types, and so no type mismatches to avoid and little typing overhead![4]

## Examples – Thunking

An example thunking function is shown in listing 4. Using a similar technique to the above, our otherwise-complicated Thunk function can now be simplified to one with just two parameters – a function number and a ParmArray containing the required parameters. The listing shows how the first of the calls below will have its parameters checked against those that the first function, fn1, requires, and if they all match, the function will be called. The parameter types are packaged up automatically and sent along with the values.

```
Thunk(1,
   (ParmArray(), "First Parameter", 2, 5));

Thunk(2,
   (ParmArray(), 1, 3.14, "Calls Fn 2", 1));
```

## Examples – IDispatch

The MFC helper class for IDispatch interfaces is called COleDispatchDriver. Its InvokeHelper() method is usually called with code such as:

```
BYTE parms[] = {VT_I4, VT_BOOL, ...};
x.InvokeHelper(id, type, retType,
   &ret, parms, a, b, c, etc);
```

In this case, `id` is the ordinal number of the method you wish to call, and `type` is the type of the call (e.g. get/put/method). `retType` is the return type, `ret` is a type-unsafe void * buffer for the return value, `parms` is an array of types, and the actual parmameters themselves are added at the end. Using the ParmArray class, a wrapper function could be created that deals with the parameter and return value types automatically in a type-safe fashion, allowing you to write something like:

```
x.SafeInvoke(id, type, ret,
   (ParmArray(), a, b, c, etc));
```

Here we have no array of types to build, no casts on the return values, and no possible mismatches of variables and their types. The SafeInvoke() method could then parse the ParmArray parameter and call InvokeHelper() itself.

## Flexible (and even Multiple) Return Values

Using VarArg for the return type of a function allows it to return any of the types a VarArg can hold depending upon how the function is called, or even upon the time of day. Furthermore, a ParmArray return type allows a function to return any number of

[3] . (If only Bjarne's overloaded whitespace proposals had been adopted, it would have offered the possibility of a syntax familiar to LISP programmers…

[4] If the ParmArray bit bothers you, you can always call it something shorter if you use it a lot!

return types to the caller, which can then process them at its leisure. A very simple example is shown in listing 5, which can be used as follows:

```
ParmArray ret = MultiReturn(7);
// This prints: 7 49 2.64575
cout
  << endl
  << ret[0].getInt() << " "
  << ret[1].getInt() << " "
  << ret[2].getDouble();

// A function that dumps as unknown
// set of parameters to the stdout
bool Dump(const ParmArray &parms)
{
  bool ret = true;
  for (int i = 0; i < parms.size(); ++i)
  {
    const VarArg& currentParameter = parms[i];
    switch (currentParameter.type())
    {
    case VarArg::INT:
      cout << currentParameter.getInt();
      break;
    case VarArg::DOUBLE:
      cout << currentParameter.getDouble();
      break;
    case VarArg::PCHAR:
    case VarArg::CPCHAR:
      cout << currentParameter.getPChar();
      break;
    default:
      ret = false;
      cout << "(invalid parameter to Dump)";
      break;
    }
  }
  return ret;
}
```

**Listing 3 – A simple implementation of a Dump function. To demonstrate the principle, it simply loops over the ParmArray object and dumps each entry, but in general could be opening files, making OS calls, etc.**

```
// The first of a set of functions
void fn1(char *msg, int start, int finish)
{
  for (int i = start; i < finish; ++i)
    cout
      << "Loop " << i << " - " << msg << endl;
}

// Thunk function calls a number of functions,
// each with a (different, but) well-defined
// signature
bool Thunk(int fn, const ParmArray &parms)
{
  bool ret = true;
  switch (fn)
  {
  case 1:
    if (parms.size() == 3 &&
      parms[0].type() == VarArg::PCHAR &&
      parms[1].type() == VarArg::INT &&
      parms[2].type() == VarArg::INT)
      fn1(
        parms[0].getPChar(),
        parms[1].getInt(),
        parms[2].getInt());
    else
      ret = false;
    break;
  case 2:
    // etc...
  default:
    ret = false;
  }
  return ret;
}
```

**Listing 4 – An example of a thunking wrapper function. The function simply passes control, and the parameters from the intelligent parameter list, to one of a set of functions (only one shown), depending upon the function number specified.**

## Safety issues

The example VarArg class contains some pointer types. Care should obviously be taken concerning the ownership of the underlying data of such pointers. This is especially true for return types were it might be easy to return a pointer to a local variable or a buffer that is deleted before the function returns. I have taken the decision that these VarArg objects never own these underlying objects. Similarly, if a pointer to an allocated buffer is returned, care must be taken to ensure the caller knows of its responsibility to delete the pointer once it has finished with it. Of course, many of these issues are no different from the case with a regular function returning a pointer type. A final warning should be given about the current lack of copy ctor and assignment operator. If other value types are added, you should make sure that the correct copy semantics and ownership rules are preserved.

Unions are not able to contain UDTs with ctors or dtors, because, of course, the compiler would not be able to decide when to add calls to these functions in the code. It is not even possible to work around this by, for example, checking the type in the VarArg dtor and calling the appropriate dtor on the union member explicitly. What you can do, of course, is to add pointers to UDTs in the union. As long as we decide that VarArg objects will never own the contained data, we can just use a pointer to the original UDT in the union, even if we hide this pointer

implementation from clients. Alternatively, if we change our stance on ownership, the overloaded conversion ctor of VarArg could allocate a new object copy of the UDT, with the dtor deleting the allocated resource at the end. I have used C pointer types for simplicity and speed, but reference counted strings could be safer and almost as efficient.

```
// A function that returns more than one value
// Note - Ownership details need to be worked
// out for pointer return values...
ParmArray MultiReturn(int i)
{
  // return the number, its square and
  //its square root
  return
    ParmArray(), i, i * i, sqrt(double(i));
}
```

**Listing 5 – A simple example of a function that returns more than one value, packaged in a ParmArray.**

### Summary

The VarArg and ParmArray classes provide a convenient and type-safe way to wrap up and communicate a variable array of variably-typed parameters, either for a function call or return value. The syntax is not unfamiliar, and it can be useful in a number of fairly common cases where a single function needs to offer a number of different function signatures to its clients. There are a few issues such as ownership and the availability of types that should be considered when implementing and using this class, but it offers a number of safety improvements over the traditional <varargs.h> alternative. Finally, if anyone can help to improve the usage syntax, then please let me know.

*Richard Blundell*
*RichardBlundell@dial.pipex.com*

# Patterns in C++

## Exploring Patterns: Part 1
## by Francis Glassborow

One can roughly divide the programmers of the World into three main groups; those that think patterns are about wallpaper, those that know they are about software design but haven't the vaguest idea about what that means, and those that litter their communications with patterns jargon. There is a small additional group of people who actually know what patterns are and how they can help with their work.

I do not belong to any of these. In this series I am going to attempt to join the latter group by writing about the subject. Hopefully, as I expose my ignorance the more expert readers will seize on the opportunity to correct me in the following issue of Overload.

Before I start, I should make clear that I deeply regret the way that the terminology for software patterns was cut and pasted from that used by Alexander in his books on architecture. The result has been pure jargon that makes the subject much harder for the newcomer. For example the idea of forces is entirely natural in the context of building architecture but, in my opinion, artificial when applied to software. Software experts too often fail to make the effort to get exactly the right word for a context (we see it all the time in the identifiers that programmers use). Naming things is hard but knowing something's 'true name' gives you a power over it that an artificial name fails to provide. For example the use of the term pointer constant or pointer value (frequently abbreviated to 'pointer') in C/C++ makes it much harder to talk about addresses. Compare:

| | |
|---|---|
| An `int *` stores a pointer to an `int` variable. | An `int *` provides storage for the address of storage for an `int` value. |
| A `long` contains a `long` value. | A `long` provides storage for a `long` integer value. |

The expert is probably happy with the entries on the left, yet the novice is often completely confused by these.

I am going to attempt to write about patterns without the jargon. You will not read words like 'forces' and 'collaborators' (a word with dark undertones for those of older generations in much of Europe) in my contributions. Nor should they turn up in anything you write for those outside the 'inner circle'.

## Patterns & Anti-Patterns

The concept of patterns and anti-patterns is the encapsulation of experience so that newcomers can benefit without having to re-invent the wheel. They can be found at all levels of software design and implementation. There are small, low-level patterns that are probably more familiar as idioms. These are often language specific. For example there is an idiom in C (originally identified by Andy Koenig in 'C Traps and Pitfalls') concerned with iterating over a known number of items:

```
for (i=0; i<n; i++)
```

If you want to do something exactly `n` times, that is the way to do it in C. Sure, there are other ways you could write it but experience shows that all the alternatives cost more either at implementation time in getting the count correct or later when you have to find the bug that getting it wrong introduced.

There is another similar idiom used extensively by the STL in C++:

```
for(iterator i = start; i != finish; i++)
```

Note the different test used in the second clause of the `for` statement. It is not a mistake but essential to the way that we wish to generalise the concept of a container in C++. Successive values of an iterator need not be sequential. There is a difference between iterating and counting.

Now it is sometimes possible to invert the earlier idiom with:

```
for( i = n; i > 0; i--)
```

However you must be careful about even such a small change to the idiom. For example, what if the body of the for-loop contains `array[i] = 0;` ? Not only will the second version do something different, it will also exhibit undefined behaviour if `array` contains only `n` elements (the notorious 'one beyond the end' problem (almost an anti-idiom.) With understanding you can write the exact reverse with:

```
for (i = n-1; i >= 0; i--)
```

and sometimes you may need to (or else reorganise your work to make it unnecessary)

However you cannot, in most circumstances, rewrite the C++ idiom as:

```
for(iterator i = finish –1; i != start;
i--)
```

In general you will need a different kind of iterator and different values for the range if you want to work through the items in reverse order.

To give you some idea about anti-patterns let's have a quick look at an 'anti-idiom'.

```
switch (i)
{
 case 1: /* do something */
  break;
 case 2: /* something else */
  break;
 case 3 /* another something */
  // etc
}
```

The use of 1,2,3… is unwise (i.e. experience shows that it leads to errors). We need to use named values.

## Distilled Wisdom

There are two important things to extract from the above. First is that (anti-) patterns are based on experience. They are not things that are invented. If someone tells you they have found a new pattern ask them where they have seen it in use. Unless they can specify at least two disparate examples of its use then they do not have a right to claim it as a pattern. The clue to pattern discovery is when you look at a design solution, implementation or whatever and get a feeling

of déjâ vu. To complete the claim you have to find at least two previous places that the solution was used and then summarise both the problem and the general solution.

The second important element is that of understanding. Thoughtless or ignorant use of a pattern gets you nowhere but trouble. Indeed much legacy C++ code is littered with conventional solutions that do not deserve to be elevated to idioms because they are rarely the best solution. As a simple example, take the self-assignment problem (writing an `operator=()` overload) that is familiar to all experienced C++ programmers (as Herbert Schildt appears to be unfamiliar with it you can reach your own conclusion as to the value of his writings.)

The traditional C++ 'idiom' is to start the body of the implementation function with:

```
if (this == &rhs) return * this;
// process assignment
return *this;
```

or with

```
if(this != &rhs)
{
  // process assignment
}
return *this;
```

Certainly if you need to write an `operator=()` for a user defined type you need to protect against self-assignment (but you might also need to protect against overlapping assignment). However the above solutions do not work well in the context of exceptions. We usually want to provide 'commit or roll-back' semantics. In other words if an exception is thrown during the process of assignment we want to restore the `lhs` to its state prior to the attempted assignment. For this we need a different idiom whose pseudo-code looks like:

```
create temporary holders for the new lhs
elements
attach copies of the rhs elements
catch exceptions
discard lhs elements
attach new elements from their temporary
holders
return the object
```

It happens that this new idiom usually works better than the old one even if there is no possibility of an exception being thrown. Inexperienced programmers are much better off with the new idiom. Experts will want both because they will realise that there are (rare) occasions where the former has advantages (lower usage of memory) that are vital to the specific requirements of a problem.

Distilled wisdom must be based on something to distil as well as understanding of what it teaches. Learning patterns by rote will do almost nothing for your progress as a programmer/software developer.

## The Singleton Pattern

I was motivated to write this series by a question posed by a correspondent in C Vu. This caused me to pull my copy of 'Design Patterns' (Gamma et al.) from my bookshelf and do a little study. What I found disturbed me. Read on and see if you agree.

The Singleton Pattern is about handling the problem of something that is unique in a context. I first came across this problem in about 1991 when I was presented with the requirement to write a Screen class that would only allow a single screen object within an application. The problem proposer's solution deeply disturbed me. It was to track the number of instances of screen objects and abort the program if it ever exceeded one. I hope you all agree that that was not an acceptable solution.

The solution that I came up with at that time was 'the class is the object'. My implementation went something like:

```
class Screen
{
  // inhibit object creation
  Screen();
  // do not forget the copy constructor
  Screen(Screen &);
  // provide class based data
  // static Screen data structures
public:
  static void changeColour(Colour);
  // etc.
}
```

I would guess that this is not too bad from a relative novice. I think that today I might use a nested class for the data and write something along the lines of:

```
class Screen
{
  // inhibit object creation
  Screen();
  // do not forget the copy constructor
  Screen(Screen &);
  struct ScreenData
  {
    //Screen data structures
    //functionality
  };
  static ScreenData * sd=0;
public:
  static void changeColour(Colour);
  // etc.
}
```

Arranging that each of the `public static` member functions checked `sd` and called for a default initialisation if it was still a null pointer.

This is definitely a viable implementation of the Singleton Pattern. You may be curious as to my declaration of a copy constructor, but without it there is a very nasty trap waiting to bite. Revolting though it may be the following is entirely valid unless the copy constructor has been inhibited:

```
Screen trap = trap;
```

It creates `trap` as a copy of itself. This is entirely an artefact of the C++ grammar (required for compatibility with C) and results in an object being created ab initio as a copy of itself. If you are interested the following is safe and generates a compile time error unless an earlier object with the same name already exists (which it cannot in this case):

```
Screen trap(trap);
```

Which is one reason that I recommend that programmers use the latter form of declaration.

If you look at the sample code in Design Patterns you will see that the authors forgot about the need to inhibit the copy constructor so those that blindly copy their model without a good understanding will write defective code. The book is about design patterns, not about C++ so the error is excusable but it does highlight the need for language specific skills when using such references.

I have another bone to pick with the authors. I really do not like the use of pointers for this kind of code. For those that do not have the book to hand, they provide the following example:

```
class Singleton
{
public:
  static Singleton * Instance();
protected:
  Singleton();
private:
  static Singleton* _instance;
};
```

Of course the reader must do an awful lot of reading between the lines (for example the skeleton is stateless – has no data – and functionless – there are no member functions) but my objection is that `Instance()` returns a `Singleton*`. Surely this should be `Singleton&`. Of course the coding technique being used requires that the `static` data member be a pointer because you cannot have null references. But `Instance()` can terminate with `return *_instance`. Which reminds me, the authors have just invaded the implementor's namespace by using a leading underscore in an identifier.

Now with all the above caveats, the idea as exemplified in Design Patterns is fine for an object of a unique type. That is an object that will only ever exist once in any form throughout a program. However we need to tackle variants on this theme. A class that is tracking the number of objects of its type currently in existence has a counter but this is not a pure singleton. While each class only contains a single counter the concept of counting will be implemented many times. We need something more sophisticated to deal with this problem. Let me digress once again to another related problem.

Screen was an example of the type of real world object that needs unique representation within an application. Wherever you have a screen object in your program it must be the

same screen (which is one motive for making such objects global). What about other resources that need unique handlers? For example, how should I manage a serial port? Now a program may have access to several serial ports, but each port needs a single handler otherwise you have the potential for some nasty collisions.

Try the following:

Write a class (as object) for a generic serial port. Now you can use templates to specialise that class for specific serial ports. Something like:

```
template<int address, int interrupt>
class SerialPort : GenericSerialPort
{
public:
  static void write(byte)
  {
    GenericSerialPort::
      write(byte, address, interrupt);
  }
  static byte read()
  {
    return GenericSerialPort::
      read(address, interrupt);
  }
};
```

The functionality is provided by forwarding to the generic base class member functions that would normally be `protected` so that the base class cannot be abused (the reason for using a base class is to provide the implementation code only once, without it each template instance would generate its own code). Now this is far from being a fully worked out solution but the approach is important. We have a base class without any public functionality, and we have derived template classes that implement the object as a singleton. The advantage is that each serial port is uniquely identified (by its address and interrupt) and the C++ implementation will resolve multiple uses down to the same object.

Now let me backtrack to the problem of the counted class requirement for a counter. Do you see any similarity with serial ports? Well each counter is specific to a class. We could implement the concept of a counter once as a base class (with all its functionality kept protected because it is not intended for general use). Then we can derive templates with the template parameter being the class for which it is to be a counter.

Now I am not advocating this as the most appropriate mechanism for providing counting within a class (indeed I tend to think that providing a counter is so basic that encapsulating the mechanism in a class is overkill) but what I am trying to show is that template classes derived from base classes (this mechanism minimises code bloat) can do much to support the Singleton pattern.

If you think about what I have written you will see that the Singleton Pattern has a tendency to turn up in places where the objects lifetime will naturally be co-extensive with the program. Now this results in another problem, one that it shares with other global objects. Singleton objects must be initialised. It is not always possible to ensure that such initialisation is independent of other objects. That is the reason that the Design Patterns book provides them via a pointer. Null pointers can be detected by code and appropriate action taken (initialise or throw an exception). The concept of lazy, or just in time initialisation is one that is desirable for Singletons and other global objects. This is particularly the case in C++ where we have a problem of the order of initialisation of global objects that are declared in different files.

Fundamentally there are two available idioms. One is the mechanism of pointer and initialisation at point of first use suggested in Design Patterns. This is clearly a good solution for genuine Singletons (program objects that are essentially unique (Screen) or uniquely bound to a real world object (serial port) or program concept (counter for a counted class) because it makes sense to encapsulate all the data and functionality in a class. To summarise this solution:

First design a class that provides the data structures and functionality of the object. Now 'hide' the constructors from the outside world by making them `protected` (experience shows that this is better than making them `private`).

Now if the object is genuinely unique add a `static` pointer to an object of class type and provide a `static` access function that returns a pointer (or better a reference) to the object. This access function can check that the object has been created and create it if necessary. If you want to provide initialisation data you will probably need `MakeSingleton()` `static` member function and arrange that the access function either calls this with default data or throws an exception if the program tries to access a yet to be created object.

If the Singleton is not genuinely unique but just an instance bound to an object that must have only one representation within the program then deriving a template from your base class may be more effective.

However all this is not the end of the subject because there are other objects that may need initialisation just in time. For example it seems to me to be pretty heavy to take an item that is naturally an `int` and create an entire Singleton style framework purely to ensure that you do not run into difficulties because of the C++ order of initialisation problem. This leads to an alternative mechanism for lazy initialisation by using a function and `static` local data item. For example, the idiom replaces

```
double x = sin(y);
```

with:

```
double & x(
{
  static double value = sin(y);
  return value;
}
```

and in the program you substitute `x()` for `x`. Whether you choose to `inline` this function depends on the degree to which you trust modern compilers to handle `static` data within `inline` functions.

Notice that this solution doesn't eliminate the problem of circular definitions, though a good compiler might issue a warning (which it could anyway even without the use of this idiom). For example nothing apart from re-coding will resolve:

```
double & y(
{
  static double value = sin(x());
  return value;
}

double & x(
{
  static double value = sin(y());
  return value;
}
```

though writing `sin(x())` might just jog your thinking into checking for circularity.

If you are using a global (or class `static`) object of a type that will be used elsewhere in your program I think it is worth considering this idiom as a solution to the order of initialisation problem.

Now let me sit back and wait for the avalanche of objections, corrections and doubt. Perhaps at the end we all understand this area that much better.

Next time I'll tackle another pattern.

*Francis Glassborow*
*francis@robinton.demon.co.uk*

# Whiteboard

# Structured Analysis: OOD's older brother?

By Alan Bellingham

## Caveat

My description of the Structured Analysis process is somewhat simplified, and possibly terminologically suspect as a result. This methodology, which was promulgated by Ed Yourdon's company Yourdon Inc., is described in *Structured Analysis and System Specification* by *Tom DeMarco* [1], and I will refer to it as Yourdon-DeMarco. However, I trust that the concepts are still understandable.

## Let me tell you a story.

Once upon a time, when King COBOL ruled the land and the Millennium was something that programmers were putting aside as a retirement bonus, it was realised that program design was a Good Idea. So the King called his trusty advisors together, and they pondered the problem, and they came up with a simple set of ideas, which they told the King would solve all problems with programming from henceforth. Chief among the advisors were Yourdon and De Marco, who said "All programs consist of data and processes, and all we need to do is consider what the data is, and what the processes are, and all will be clear."

And Knuth concurred, saying "Data + Algorithms = Programs".

And the flowchart was worshipped, and the Data Flow Diagram deified, and all was well with the kingdom henceforth.

## And they all lived happily ever after

That's enough fairy tale for now. What were the issues that lead to the Yordon-DeMarco design methodology, and what was that methodology?

Firstly, in the business environment, there is one important thing to remember about COBOL - the data files did not necessarily *describe* the data. For those of you used to using SQL or xBase, this is a strange idea, but effectively the data in the databases was raw and unstructured, and it was the purpose of the *program* to describe what the real structure was.

Because the data itself was effectively type-less, it became the responsibility of the File Section of a COBOL program to describe the data. Inevitably, this ubiquitous component became a design consideration.

(Another way to say this is that the data was in the files, but the meta-data was in the program. These days, this attitude seems rather primitive for database handling, but it still applies to many other data structures. For instance, to decode a Tagged Image File Format (TIFF) file, I need to write code that implicitly understands where to find the Image Header, what sub-fields this contains, etcetera, etcetera. This structure is described in a specification I obtained from Hewlett Packard many years ago.)

So, the concept of the **Data Dictionary** was formed, though it went far further than just a table description.

Secondly, a process should do one thing, and do it well. This meant that the process took as input one or more data items, and produced one or more data item as output. All of these data items were precisely specified in the Data Dictionary.

Thirdly, data only existed either in transit between processes, or residing in dumb storage (e.g. in a database of some sort).

Fourthly, state was unimportant. This is a corollary of the above.

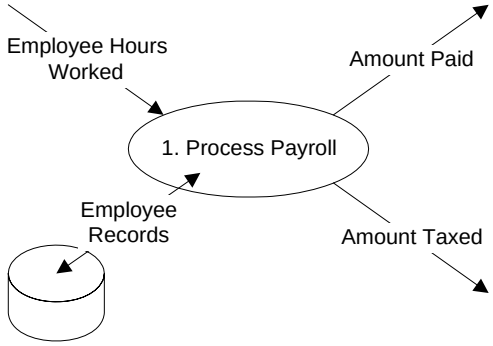
Fifthly, control was unimportant. Yes, something started the processes off, but the idea that this methodology was based on was that of the steady state data flow, even if that flow comprised a single transaction.

Sixthly, whatever was produced had to be understandable by the customer, because only then could the customer agree to the analysis,

and was therefore less likely to be surprised by the functionality of the delivered product.

### Putting it together - the Data Flow Diagram

Yourdon-DeMarco is very much a top-down methodology. The system as a whole has certain inputs and certain outputs, and can be expressed as a single bubble that does something to the inputs to produce the outputs:




The initial step would be to draw out the single bubble for the system, and to draw the inputs and outputs in. Note that the arrows are data items, and are named using nouns, while the bubble itself is a process, and therefore is named using a verb. This naming method will be used throughout. This is the top level **Data Flow Diagram**, or DFD for short.

Now that we have the initial idea of the data flows, we need to define exactly what that data comprises before we may proceed any further. Therefore, it is time to look at the **Data Dictionary**.

### The Data Dictionary

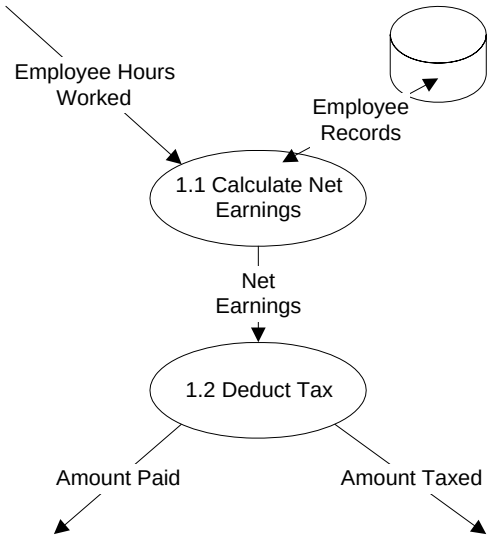
Every single data flow in the system has to be documented as to its exact structure. Now, this structure might be simple:

```
EmployeeHoursWorked= EmployeeNumber +
                     HourCount
EmployeeNumber     = 1{digit}6
HourCount          = 1{digit}3

EmployeeRecords:   = 1{EmployeeRecord}n
EmployeeRecord:    = EmployeeNumber +
                     EmployeeName +
                     HourlyRate

AmountTaxed:       = EmployeeMonetary
AmountPaid:        = EmployeeMonetary
NetEarnings:       = EmployeeMonetary
EmployeeMonetary:  = EmployeeNumber +
                     MonetaryAmount
MonetaryAmount:    = 1{digit}12
```

or it might be arbitrarily complicated. Note that the notation used here is similar to the Backus-Naur specification of a language - so that 1{digit}3 means 1 to 3 sub-components named ‘digit’. (Actually, there are a number of possible notations - but this one is simple, compact and relatively unambiguous.) Ultimately, one should be able to start with a high-level name and be able to use the Data Dictionary alone to completely determine its possible content.

### Sub diagrams

The initial diagram isn’t terribly useful in its own right. This is all right, though, because the next stage is to decompose that initial diagram into sub-diagrams:




You should notice that the flows in and out of this diagram are exactly the same as those into and out of the previous bubble, but that there is now an extra flow within. Of course, this extra flow needs documentation.

This process may be applied recursively to bubbles 1.1 and 1.2, each yielding an extra diagram of from two to seven bubbles (a rule of thumb that, like the rule that a function should fit on a page, may be broken if you need to), until the process obviously cannot be taken any further. At this stage, you should

have reached the primitive processes themselves.

Ultimately, you have a complete set of diagrams, with each diagram having the same index number as the bubble in the next diagram up that it's explaining. Oh, and in my experience, the diagrams were usually drawn in pencil, since you needed the ability to easily amend them.

## Structured English

Thus far, nothing has really specified what these processes actually do. The names of the processes and data flows are, we hope, meaningful (and there is a general rule that 'thing', 'data' and suchlike tags are *not* meaningful - 'Process payroll' isn't that good a term, either), but nothing has been said of the **how**. The final stage of the design process is to specify exactly what occurs within a primitive, and this is the point where iteration, decision and so forth may take place.

This **how** is represented in a pseudo-code known as **Structured English**. This is a form of language that is precise enough to be unambiguous, while still being sufficiently free of jargon to be understood by the customer for whom the design is being done.

In this case, I will assume that bubble 1.1 is a primitive, and show a possible Structured English result:

```
Take EmployeeNumber from EmployeeHoursWorked
For each EmployeeRecord in EmployeeRecords do
if EmployeeNumber matches EmployeeRecord then
 NetEarnings is HourCount times HourlyRate
```

This should be sufficiently clear to be agreed or rejected by the customer (for instance, the customer would probably realise at this stage that nothing had been said about overtime rates), and yet it's also precise enough to be translated into code with the minimum possibility of confusion.

## So?

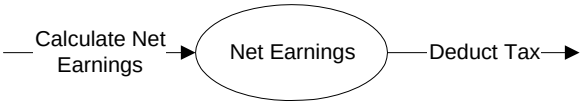
So what does all this procedural design process have with the brave new world of Object Oriented Design, I hear you ask?

At first sight, relatively little. I last used this methodology for an entire system some 8 years ago because, for a system where OOD fits well, it is almost completely inappropriate. Similarly, I hardly ever use flow charts.

Well, students of the process of creativity will tell you that there are several patterns to creativity itself, and that one of them is to take an existing idea, and try to invert it. I won't say that it's what actually happened in this case, but let's see what exactly we can do.

## Watch the rabbit

Let us take another look at the DFD, and see what happens if we do a transformation. What can we try?




Well, one characteristic of the DFD is that processes are represented by bubbles, and data by lines that connect them. Can we invert this?

Yes, we can:

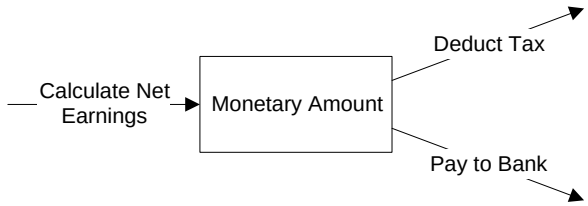
Let's pretend that the NetEarnings is a data structure. Well, actually, it is, isn't it? Just look at the Data Dictionary and we can see that it's composed from a very precise set of members.

We also know that there are two processes that know about this structure.

Hey presto, we have an object with data and methods. This might be somewhat artificial in this case - as what's the point of such a simple object. However, if we look a little further, it should be obvious that AmountPaid and AmountTaxed are very similar structures. In fact, they're really the same structure, just used in a different place. If I'd used more detail (perhaps with data flows for transferring the amount to a bank account), more and more processes would have been apparent.

Take this far enough, and we end up with a mass of bubbles, all interacting by calling each others' methods (or messaging each other according to some terminologies).

This result is an Object Interaction Diagram - and it's undeniably a different beast from the DFD that we started with:




Compare this with the Message Connection notation shown in O*bject-Oriented Design* by *Coad and Yourdon*[2] (and first formulated in *Object Oriented Analysis*[3] by the same authors).

So, what we've ended up with is OOD rather than DFD. Instead of considering processes first, and how they act on data, we start first with the data, and then consider the processes that act upon it. As a result, we can *now* consider state - because the object necessarily contains state, and that is because it persists between messages. Interestingly though, we lose the ability to easily consider data creation - but then, any design process needs to consider a number of different methods

## Conclusion

There are few new things under the sun, and this applies to programming, too. Sometimes however, by conceptually inverting an existing technique, a new one may be seen. Contrariwise, the existence of newer techniques doesn't mean that older ones should be completely rejected - just as a flow chart may still be useful on occasion, so may Structured Analysis.

Oh, and a warning: be careful of what you say if you ever get into conversation with Francis Glassborow, because he may want an article on it.

*Alan Bellingham*
*alanb@episys.com*

## References

[1] Structured Analysis and System Specification by Tom DeMarco (Prentice-Hall 1979 ISBN: 0-13-854380-1)

[2] Object-Oriented Design by Peter Coad and Ed Yourdon (Prentice-Hall 1991 ISBN: 0-13-630070-7)

[3] Object-Oriented Analysis by Peter Coad and Ed Yourdon (Prentice-Hall 1990 ISBN: 0-13-629981-4)

# Object Design and Implementation by The Harpist

I was trying to think of a suitable topic to use to explore some ideas for the design and implementation of object types, as opposed to attribute types (value based types), when Francis forwarded an extract from an email dialogue with Paul Collings with his permission for me to do a code review and publish the results.

Actually I am going to range rather wider than an ordinary code review and I hope that the various experts will chip in their bits. One problem with being a local expert is that you are expected to spend time in educating colleagues rather than in attending further training for oneself. This means that almost all I know I have acquired by reading and discussion over the odd pint. I am sure that many readers are in a similar position so that when I reveal a blind-spot I am sure it will be shared with quite a few others.

## Paul writes

*I would also be grateful if you would look at the following which works quite happily as a single file, but when split into multiple units it all starts to fall to pieces as I know very little about the linker operation of a compiler.*

I know that Francis has already suggested that time would be better spent reworking the code from scratch rather than trying to locate

the subtle interactions that are causing problems. I would reiterate this in the strongest terms. I would go further and suggest that any time you develop even a small program as a single file you will be laying up maintenance problems for the future.

Paul set out to create a 'Hotel' type. His view of a hotel was that it was an aggregate of rooms for hire: bedrooms and function rooms. That is fine, as far as it goes, but if we are interested in reuse we should recognise that time could be invested in providing a wider range of components for our hotels such as those reserved to staff (utility rooms, kitchens, corridors etc.) and public rooms (dining, bars, lounges etc.).

Paul added a second type of component to cater for the concept that a hotel might have movable presentation equipment for use in its function rooms. Again this is only one example of movable equipment and we might usefully generalise the concept. Let me look at Paul's provision of a type for presentation equipment and see what we can learn from that.

```
#if !defined PREQUIP_HPP
#define PREQUIP_HPP
// although this has no definitions
objects can be instantiated
class PresentationEquipment
{};
#endif
```

As provided this is a stateless (no data) functionless class that will have the four standard compiler generated functions (constructor, copy constructor, copy assignment, destructor). It looks like a placeholder for something more. In other words it is the class equivalent of a stub function so I guess we might call it a stub-class. Let us ignore, for the time being, the question as to whether it is the most primitive type (highest abstraction?) and focus on the fact that realistically it will be a base class for a variety of different types of presentation equipment.

Do you see any problem with `Presentation Equipment` being a base class? Where else have you come across stateless, functionless classes?

One big use for these is to provide exception objects. We often want to distinguish between different types of exceptions that can be thrown in a specific context. Sometimes we want to handle the whole family of exceptions the same way, at other times we want to be more specific. Often the only relevant item is the name of the exception so we get something like this:

```
namespace HarpistExamples {
class Mytype
{
  // private interface
public:
  // public interfacre
  class Exception {};
  class Problem1 : public Exception{};
  class Problem2 : public Exception{};
  class Variant1: public Problem1 {};
  class Variant2: public Problem1 {};
  // etc.
};
}
```

Now the user of the type can write:

```
using HarpistExamples::Mytype;
// whatever
try {
  // normal actions
}
catch (Mytype::Variant1) {
  // specific action
}
catch (Mytype::Variant2) {
  // specific action
}
catch (Mytype::Problem1) {
  // handle Problem1 generically
}
catch (Mytype::Problem2) {
  // handle
}
catch (Mytype::Exception& ex) {
  // handle all other Mytype exceptions
}
```

The user does not need to know if there are other types of exception that can be specifically generated by `Mytype` objects. The owner of `Mytype` is free to add other exception types confident that the experienced user will be providing some form of handler. However there is a very small problem, none of the derived types can have any dynamic resources. That is a very small problem because writing exception objects that use dynamic resources is asking for

trouble (the potential for a double exception, one thrown during the process of passing the exception object). That, by the way, is a good reason for not using the standard `string` type in an exception type. Let me elucidate a little further.

When an exception is thrown the exception object is copied to the handler. Think about it and you will realise that it must be this way. By the time you get to the handler the stack has been unwound and the original object is gone. This places a constraint on the types that can be thrown, they must have publicly available copy constructors (and destructors). You might ask what happens in the last catch clause above where the exception is caught as a reference. A temporary has to be created for the copy so that it can be bound to the reference. If you do not catch by reference only the base class will be copied. When you catch by reference the whole of the original is copied and is available for processing and possible rethrowing (`throw;` in the above[1]) if you want to allow more complete handling higher up your code.

The possibility of catching by reference should alert you to the critical missing element in the base class without which you will not get the behaviour you expect (nor will you be able to use a `dynamic_cast` to access the dynamic type functionality). Stateless base classes need virtual destructors!

Oddly, providing this may be practically free. All class objects must have a non-zero size and in many cases memory alignment considerations result in the minimum size being that of a pointer which is all the overhead that polymorphic types impose on their instances. In other words you recycle the unused memory of a stateless type by storing a virtual function table pointer in it.

The benefit is that you have upgraded your base type to one supporting Run Time Type Information which can be used by `dynamic_cast` and whose derived types can have dynamic resources without leaking.

If you want exception types to encapsulate data, I think a good case can be made for providing that data as `class` data (`static` members of the class) rather than as object data. However this is not quite as clear-cut as it might seem at first because a nested exception of the same type might over-write the data. I will leave thoughts about that for another time (or perhaps one of the expert readers might like to take it up).

## Defining PresentationEquipment

In view of the above I think that the following is a more robust definition and is less likely to fall foul of future development of the concept:

```
class PresentationEquipment
{
public:
  virtual ~PresentationEquipment()
  throw() {}
};
```

Apart from my reservations about this being the fundamental base class I think that should cover Paul's original intent. Qualifying the destructor as virtual ensures that RTTI is available if and when you elect to use it. Defining it in class means that it will be inline, which seems an eminently sensible thing to do for a destructor with an empty body for a stateless class. The only other

[1] There is an important syntactic element here that many programmers miss. If I want to rethrow the original object I must simply write `throw.` If I want to throw the local copy (whose static type is now `Mytype::Exception`) I write `throw ex;` This latter form strips off all the dynamic information because it is an absolute rule of exceptions that the copy constructor used is that for the static type of the object thrown, not its dynamic type. Locally the static type of `ex` is `Mytype::Exception`, though its dynamic type is the static type of the original thrown object. Until all this nested type information makes sense to you, you would be well advised to keep it simple. For further information read Item 12 of 'More Effective C++' by Scott Meyers (ISBN 0 201 63371 X)

thing to note is the exception specification. The `throw()` form declares that the destructor will not leak exceptions. In other words it can be safely called during the process of stack unwinding as a result of handling some other exception. At first sight you may question placing such a constraint on a function with an empty body. In general you would be correct, but virtual destructors are a special case because of their polymorphic nature. The bodies of destructors of more derived classes will not always be empty.

Another advantage of adding the exception specification is that the derived class destructors will be required to meet the same constraint. I believe this means that omitting the specification (or providing a different one) on an explicitly declared derived constructor is a compile time error. If it isn't, it should be.

### And Next

Paul's next header file is:

```
#if ! defined CUSTOMER_HPP
#define CUSTOMER_HPP

class Customer
{
  static int customerCount;
  char* name;
  char* payee;
public:
  Customer();
  ~Customer();
  char* getName();
  char* getPayee();
  static int getCustomerCount();
};

#endif
```

The first problem is that `static int customerCount.` When aiming to create a hotel object it seems reasonable to want a count of the number of customers. However a little further consideration should show that the data for that is part of the hotel object. Think what would happen if your application had two hotels. Each needs to track its own customers.

The next defect is a combination of using `char *` for the `name` and `payee` together with the lack of explicit provision of a copy constructor and copy assignment. There are two ways to fix this problem. The traditional mechanism is to declare `Customer(Customer const &)` and `Customer & operator=(Customer const &)` as `private` members of `Customer` until you decide that you want to implement copying either `public`ly or as `protected` members (for the benefit of derived classes). That way any attempted copying by a user with the consequential problem of cross-linking objects (two objects pointing to the same dynamic array) will be detected at compile time.

The second solution now that we have a reasonable `string` (`template`) `class` in the Standard C++ Library is to avoid this complication by replacing the `char *` by `string`. Even so you should ask yourself if copying customers is a valid concept in the application domain (I suspect that it isn't, but read on).

The next defect is in the return types of `getName()` and `getPayee()`. Where read access is being provided to internal data it is vital that the return type is `const` qualified. Otherwise you have provided access to critical features of the `private` interface. For example, consider what would happen if you had released the above class definition and then realised that the internal data was better implemented as `string`s. You could not make the change because some user might have utilised the fact that your access functions allowed access to the raw data.

Putting this all together we get:

```
class Customer
{
  string name;
  string payee;
  Customer(Customer const &);
  Customer & operator=(Customer const &);
public:
  Customer();
  ~Customer() throw();
  string const & getName()const throw();
  string const & getPayee()const throw();
};
```

Note that as well as the change in the return type of the read access functions I have added `const` qualification (reading data should not change it) and an empty exception specification (reading data should not cause an exception). The former is certainly necessary; the latter might not always be the case. If you do something in the definition of either function that might result in an exception leaking out of the function call good compilers will generate an error (bad ones will just ignore it as diagnostics are not required here).

However, I still have not finished with this class. The idea seems to be that you can have a customer who is actually hiring facilities and someone else who is paying for them. That is perfectly reasonable (think of the standard business type booking, the direct customer is one of the employees while the payee is a company). The question that springs to my mind is what happens when the same person books a function room and a bedroom. Even more likely is that the same person is paying for several rooms. We need to encapsulate that data in such a way that the same sub-object can exist in more than one place. I think we need something like:

```
class UserName;
class PayeeName;
class Customer
{
  UserName & name;
  PayeeName & payee;
  Customer & operator=(Customer const &);
public:
  Customer
      (UserName & uid, PayeeName & pid);
  Customer(Customer const &);
  ~Customer() throw();
  UserName const& getName()const throw();
  PayeeName const& getPayee()const
throw();
};
```

Now it makes perfectly good sense to support cloning (copy constructor) because the same details might apply to more than one room. On the other hand we cannot easily support assignment because C++ does not allow you to rebind a reference. This version also provides a problem if we should ever want a collection of `Customer` because there is no default constructor. So it seems that using a reference for `name` and `payee` doesn't work as well as we might want. If we need to 'rebind' data then we have to resort to pointers and we get:

```
class CustomerRecord
{
  UserName * name;
  PayeeName * payee;
  static UserName un(missing);
  static PayeeName pn(missing);
public:
  explicit CustomerRecord(
        UserName * = 0, PayeeName * = 0);
  CustomerRecord(CustomerRecord const &);
  ~CustomerRecord() throw();
  CustomerRecord & operator=
                (CustomerRecord const &);
  UserName const & getUserID()const
                                    throw();
  PayeeName const & getPayeeID()const
                                    throw();
  CustomerRecord & changeUserID(
                        UserName const &);
  CustomerRecord &
        changePayeeID(PayeeName const &);
};
```

Now we can have a default constructor that sets both pointers to null. Unfortunately that provides a mechanism for implicit creation of `Customer` from `UserName`. Qualifying the constructor with `explicit` turns off that C++ feature. Note that the access functions can still return `const` references.

`UserName` and `PayeeName` have just been declared. In practice these type are likely to be derived from a common base. They might even be `template` classes so that the different types (individual, corporate etc.) of clients can be used. However I think I have gone far enough for this time. (And we haven't even got to rooms yet!)

Now it is your turn to pull my code apart.

*The Harpist*

## Broadvision: A lesson in application frameworks
## By Sean Corfield

This is intended to be the first in a short series of articles describing my experiences with a product called Broadvision - a C++ application framework for building electronic commerce (e-commerce) web sites.

## First Impressions

I started my current contract with IS Solutions (http://www.issolutions.co.uk) on 17 March, 1997. They used to be mainly a Digital VAR but moved into facilities management and enterprise management over time, then in recent years acquired a web design company and now most of their business is web-based. I was employed as a C++ guru along with Doug Clinton - a fellow ex-BSI C++ panel member - to act as "Senior Designer" on a major e-commerce web site.

The project would involve a lot of C++, a lot of Oracle and SQL… and a product called Broadvision One-to-One (http://www.broadvision.com) which had been 'recommended' by IS Solutions' client and, after a brief evaluation, approved by IS Solutions. Since this was the unknown technology in the project, the first thing that happened was a training course. Broadvision were in the process of setting up their UK offices so their UK support person - a French girl, Marine - was also on the training course, which was run by a chap from their Dutch office, Niek.

Unfortunately (for us or for Niek), we were to learn about version 2.5 of the product and Niek was only familiar with the prior version (2.1 if memory serves). We were all in for an interesting and informative week!

So what is Broadvision? First, let me describe its aims and then I'll explain what the product actually is. Broadvision is intended to let you sell 'product' over the web using targeted marketing. In other words, it is intended to provide personalised advertising, editorials and product offers by recording how you browse (within a Broadvision-powered web site) and some personal details about you (a user profile that you fill

in). The course was a bit of a marketing pitch too, as is so often the case with product-based training courses, and to be honest it sounded just the ticket for our project. I made notes during the course that suggested Doug and I would probably only need to make minor customisations here and there whilst the majority of our work would be integration. If this had proved to be the truth, I wouldn't be writing these articles of course.

Anyway, so what is Broadvision? Hang on, we still need some background on web applications! When you visit a dynamic web site, the first page you visit has to start a 'session' on the server side against which all your actions and choices can be logged, the session lasting until you explicitly log off the site or else it times out when it hasn't heard from you for a while (because you've gone browsing elsewhere). One of the most common dynamic web experiences that follows this pattern is an ASP (Active Server Page) site such as Microsoft's own. The ASP pages are a mixture of HTML and either VBScript or JScript. Some of the session management is explicit in the script programming and some is implicit in the ASP engine on the server. As those who have visited Microsoft's site will know, ASP relies heavily on 'cookies' - small files stored on your hard drive by the browser which contain 'lookup' information which can be passed back to the web site on request. That's how many dynamic web sites recognise you and cookies, whilst considered by some surfers to be an invasion of privacy, are generally a good way of providing a more personalised path through the mire that is the world wide web. In addition, the scripts in the pages can produce dynamically generated HTML based on information from databases, typically accessed via ODBC.

Not everyone likes cookies though and ISS's client had serious objections to them. Broadvision solves this problem by maintaining all the persistent information on the server and using complex URLs and hidden data fields in forms to help the Broadvision engine keep track of which web user is doing what at any one time. Broadvision has a number of Unix processes that manage web user sessions and provide access to databases and other resources. From the programmer's point of view, Broadvision is a set of classes that generate HTML and process GET and POST data submission operations, and also a fairly low-level API

which gives access to the underlying Oracle or Sybase databases.

The range of objects seemed broad and, at first, well-suited to the client's requirements for tracking customers, providing targeted offers and selling foreign currency, travel books, flights and holidays over the web.

## Second Impressions

Once the course was over, we set about preparing for the project proper. Our Sparc 5 workstations and Ultra server turned up, running Solaris 2.5, and we installed Oracle 7.3.2 then Broadvision. After all, it runs on Solaris 2.5... well, actually no, it specifically requires 2.5.1 so we started all over again. Great, now we are ready to run the demo applications! No, 'fraid not. Can't quite remember the exact sequence of events at this point but we did eventually get things sorted out and, having 'proved' the technology, started to design the system.

This is when things began to get interesting. Broadvision, in common with many application frameworks, makes a lot of assumptions about what you, the programmer, are going to be doing with it. It assumes, for example, that the 'product' you're going to be selling has a fairly simple structure, the sort that can be modelled by a single product table in a database. It assumes that when you pick product up off the 'shelf' and put it in your 'shopping basket' that if you pick up an identical product as well, it can just remember the quantity '2' in the basket. It also assumes that a user either visits the site as a 'guest' and doesn't buy anything or that they 'log in' right at the beginning. All these assumptions were to cause us interesting problems over the next six months.

## The Learning Curve

Having established that we had to undertake some non-trivial amount of customisation, we had to start figuring out exactly how Broadvision worked under the hood, how its class structure worked, what the API provided and in particular how it interacted with the database. The latter was critical to most of our customisation since we had to ensure that we didn't break any of the functionality that we found useful while still overriding the functionality that didn't work the way we required.

## Architecturally Speaking

Moving on to the meat of this series, I'll now look at the actual architecture of Broadvision to give you a flavour of what I was up against. In future articles I will look more closely at exactly how I progressed towards the current, live site.

Broadvision has a session manager process - the CGI program - that handles all POSTs and GETs from the user. Each operation sends a series of hidden fields that tell Broadvision what 'object' to invoke to handle the request as well as the parameters to that object. For example, a link which would look like:

```
<A HREF=hol/hlb03f.t>Destination</A>
```

in plain HTML becomes something like:

```
<BVBlockObject Dyn_SmartLink
receive_class=Dyn_SmartLinkReceive
destination=hol/hlb03f.t>Destination
</BVBlockObject>
```

in Broadvision tags. Broadvision interprets the extended HTML file, loads the 'Dyn_SmartLink' object (from a shared library), invokes various methods on it to handle the attributes ('receive_class' and 'destination') and generate the actual HTML sent back from the web browser. The processed HTML looks something like:

```
<A HREF=/cgi-
bin/bv.cgi?BV_EngineID=0.766.1.2.3.4
BV_Operation=Dyn_SmartLinkReceive&
    BV_SessionID=AFDHFRGB&
    BV_ServiceName=Mall&

form%25destination=hol%2fhlb03f%2et>Desti
nation</A>
```

In reality, the engine ID and session ID would be much more complex. When the user clicks on the link, Broadvision loads the 'Dyn_SmartLinkReceive' object and invokes various methods on it to handle the attributes (as before, except they are now prefixed with 'form%25') and the Broadvision session

manager selects the next extended HTML file (from the 'destination' attribute) and the process begins again.

Each object ultimately extends a 'Dyn_Object' base class and overrides its methods. The sequence of methods called is very rigid: to generate HTML, Broadvision calls 'prepare()' which calls 'prepare_attribute_list()' which calls 'prepare_attribute()' on each attribute given. Then it calls 'handle_attribute()' on each attribute and finally 'handle_body()'. When handling an incoming request, it calls 'receive_attribute()' on each attribute and then 'receive_body()'. You override the methods within your own objects to record and process each attribute, generate the HTML and process incoming attributes respectively.

Broadvision expects each object to process attributes and store them in named private data members. Think about it: each object performs certain identical functions in terms of dealing with name/value pairs, and yet you have to duplicate the code in every single derived object!

Needless to say, my first revision was to create generic 'receive' and 'submit' objects that inherited from 'Dyn_Object' and used a 'map<>' to store any attributes found. My own objects then inherited from these, resulting in much less duplication.

This was not a good sign! Broadvision's approach to code reuse amounted to brute force: cut'n'paste. A typical object written using this approach amounted to some 200 lines of identical code to the base class... and that was before you added your own functionality. By abstracting the common attribute handling into a base class and clearly separating 'generators' from 'receivers' my own objects amounted to about 40 lines of framework into which I could drop my own functionality.

My next problem was that I still had to override 'handle_body()' and 'receive_body()' in toto because those routines also perform certain operations that are standard and must be duplicated in every derived class. The solution to this was to change the base class function to call a private virtual which, in the base class, did nothing but in your derived class performed the necessary specialised task.

```
// base class:
void My_Receive::receive_body()
{
    // standard stuff
    specialised_receive_body();
    // more standard stuff
}
// virtual
void
My_Receive::specialised_receive_body()
{
    // do nothing
}

// derived class:
void My_DerivedReceive::
specialised_recieve_body()
{
    // do derived stuff
}
```

I hope I've given some flavour of what happens when you use a framework that isn't ideally suited to your application. Next time, I'll look at database access and flyweight classes to encapsulate some of Broadvision's raw API as well as looking deeper into subjects such as session management.

*Sean Corfield*
*sean.corfield@issolutions.co.uk*

## STL Implementations: Personal Experiences By Phil Bass

### In the Beginning

The Standard Template Library first came to my attention when I received a floppy disk along with a copy of C Vu in November 1994 (or thereabouts). It was, of course, the set of generic algorithms developed by Alexander Stepanov and Meng Lee at Hewlett-Packard. It was significant to C++ programmers because it had been proposed for inclusion in the C++ Standard Library. But for me it was much more than that.

## Revelations

Since getting my first C++ compiler some two years earlier I had been uneasy about containers. I had read articles about container classes, I had written some myself, but something wasn't quite right. Object-oriented programming was supposed to mean never having to write your own List. So, why were there so many variants of List classes? And why was it necessary to choose between efficient, robust or easy-to-use versions?

Then, along came STL. Here was an approach to containers that was general, efficient, easy to use and elegant. I couldn't wait to start using it.

## First Impressions

The first difficulty I had was understanding the documentation. It had an unfamiliar academic style and the code examples used features of C++ I had not met before. But, after some effort, I began to make sense of it.

Then there was the problem of finding a suitable compiler. At work, we were using an early version of Visual C++, which did not support templates. At home, though, I was able to experiment with my own Borland and Symantec compilers.

## The Evangelist

Some time around March 1995 (I think) the STL was accepted as part of the C++ Standard Library. It was then that I started to sing its praises at work. My colleagues were interested at first, but this quickly turned to puzzled scepticism, or worse.

A typical conversation went something like this...

*Me*: I think we should use vector<T> from the Standard Template Library.

*Colleague*: What's that?

*Me*: (trying to keep it simple) It's an array that expands as necessary when you add things to it.

*Colleague*: Is it part of MFC?

*Me*: No, it's part of the Draft C++ Standard Library.

*Colleague*: Is it provided with Visual C++?

*Me*: No, but it's not specific to any particular compiler vendor.

*Colleague*: What's wrong with the MFC containers?

*Me*: (resisting the temptation to explain exactly what was wrong with MFC containers) They only work for a limited set of data types.

*Colleague*: Yes, but we can always use "void *".

*Me*: True, but that means all sorts of unsafe type casts.

*Colleague*: (unconvinced of the strength of this argument) Are vectors efficient?

*Me*:Yes! About as efficient as you can get for an array-like container that can grow.

*Colleague*: You mean, as efficient as the MFC containers?

*Me*: (becoming irritated) No. More efficient than that.

*Colleague*: (with disbelief) Really? Why doesn't Microsoft provide the STL?

*Me*: Probably because they want to tie us all in to Windows. Perhaps because they don't have the technical expertise.

*Colleague*: (looking at me as if I'm stark staring mad) All right, then. Exactly what is it that makes the STL containers so much better than MFC's?

*Me*: (with a sigh) For a start, STL containers are accessed using iterators, which de-couples the implementation of the containers from the algorithms that use them. That means that the standard algorithms will work for all the STL containers. And because

iterators are a generalisation of pointers, the standard algorithms will work with ordinary arrays, too. Furthermore, the containers are class templates, so they can hold any object, including those particular to our applications.

*Colleague*: (whose eyes have glazed over) Templates? Are they supported in Visual C++?

*Me*: No, but we could always use another compiler.

*Colleague*: (shuffles off with a look of sheer horror on his face)

I'd blown it, of course. That colleague was now convinced that all my ideas were positively dangerous. A commercial organisation can't afford to waste time on fancy ideas nor to take risks with new and unproved technologies.

## Ahead of My Time

Eventually, we upgraded to VC++ 4.0. Templates were supported and MFC acquired some container class templates. Microsoft's Help even explained why the templates were a Good Thing.

The old arguments were raised again. By now, our software engineers were more comfortable with C++ and a little more willing to explore new language features. But there were still no vectors in Visual C++.

We looked at the Hewlett-Packard STL. It seemed to work well, but it was unsupported and it didn't have strings. A search through some magazines turned up three more STL implementations: those by Modena, Rogue Wave and ObjectSpace. The Modena library was not available in Europe. The Rogue Wave library seemed to be carrying some baggage from earlier Rogue Wave class libraries. So we ordered a trial version of the STL<ToolKit> from ObjectSpace.

## STL<ToolKit> by ObjectSpace

I was immediately impressed by the STL<ToolKit>. The software itself came on a single floppy disk which contained full source code for the STL together with over two hundred examples and a configuration program. There was also a comprehensive manual with an excellent tutorial. (For the record, this is still the best tutorial on the STL I have seen.)

The installation procedure was simple and painless. There followed a rather longer configuration phase. To see why, put yourself in the position of a library implementer. Remember, this was back in1996 when the C++ standard was in flux and some compiler vendors were tracking the standard more closely than others. The STL has always stretched compilers to the limit and ObjectSpace were forced to code round all sorts of bugs and missing language features. Even worse, new versions of compilers were coming out every few months. To have any hope of delivering a stable product, the STL<ToolKit> had to be easy to configure. ObjectSpace solved this problem with a configuration program.

The configuration program ran your compiler (from the command line) on several tens of source files. Each file tested a particular feature of the compiler. If the compilation failed the configuration program added a #define to the config.h file which was included by all the headers in the STL<ToolKit>. The #defines ensured that the STL implementation worked with your compiler. If you changed from one compiler vendor to another or upgraded from one version of compiler to another the STL<ToolKit> could be reconfigured by running the configuration program again.

Having configured the product, I built the example programs. As far as I can remember there were two failures among the 250 or so programs and neither were attributable to ObjectSpace. After a little playing we ordered a few licences for production code.

The only real problem we encountered was that debugging was sometimes tricky. On the rare occasions when we wanted to trace into the STL code we found it difficult to understand. This was partly because of the

requirements of the Standard, partly because of the many #if directives that provided configurability and partly due to undocumented macros used to help the implementers.

With hindsight, there was one other point that should be mentioned - STL<ToolKit> was an implementation of the STL containers and algorithms, not a complete implementation of the Standard Library. In particular, it did not support wide character I/O streams or locales.

## The DinkumWare Implementation

A few months later Microsoft brought out VC 4.2, which boasted a full implementation of the C++ Standard Library written by P. J. Plauger and his colleagues at DinkumWare. For us this meant three things:

1. The STL was provided (free) with our VC 4.2 licences.

2. We had a complete implementation of the Standard Library.

3. The new implementation had the best possible pedigree.

No matter how good the ObjectSpace library was, there was (sadly) an unassailable commercial case for switching to the DinkumWare offering, which we did.

The transition was not entirely painless. ObjectSpace had chosen to omit the default template parameters specifying allocators, whereas DinkumWare had chosen to make them obligatory in the standard set of header files. Microsoft provided an additional non-standard header file, <stl.h>, that enabled the allocator parameters to be omitted, but it had some obvious bugs and we decided not to use it. That meant that we had to change all our container declarations. Either we had to use long-winded type names or we had to add lots of typedefs. It's surprising how irritating this turned out to be.

The code for the new library was also a lot more difficult to understand than the ObjectSpace code. This was partly because it was a full implementation, partly because it used rather terse variable names and partly because the layout was cramped and unfamiliar.

Although I can't remember the details, we also found a bug or two. So, on balance, I think it would have been better to have waited for a later release before switching to the DinkumWare Library. The experience did bring home to me, though, that implementing the C++ Standard Library is an enormous task for a small company. That first release wasn't perfect, but it was a tremendous achievement.

We used this version of the Standard Library for the "core" of our application. The User Interface and "infrastructure" used MFC extensively and the developers on those teams preferred to stick with the MFC containers. Whether the developers were ever really convinced of the merits of the STL I'm not sure because I moved on to other things.

## Pastures New

Last year (1997) I joined another company. The tool set, however, remained the same. (Visual C++ 4.2 on Windows NT 4.0.) The mind-set was similar, too. The team was using MFC, including MFC container classes. The project was only just beginning to generate code, so introducing STL containers was not unthinkable. And, as it turned out, there was a good reason for changing direction. The software was planned to run partly under a real-time operating system for which MFC was not available. So for this part, at least, standard containers made sense.

Once again, there were some concerns about the STL and the Standard Library in general. It was (and still is) regarded as difficult to learn and difficult to use. Particularly scathing comments have been made about Microsoft's documentation and the number of warnings generated by the DinkumWare library at maximum warning level. For comparison, at warning level 4, a trivial program generated about 500 warnings using the DinkumWare headers; this went down to 2 warnings using the ObjectSpace headers!

The DinkumWare library also causes memory leaks under some circumstances.

As a result of this experience we planned to abandon the DinkumWare library in favour of a revised version of the STL<ToolKit> now called Standards<ToolKit>. But then other events overtook us. First, we decided that we probably don't need to use a real-time operating system - NT will do what we need. Also, to make better use of our existing Delphi expertise we decided to switch to Borland's C++ Builder. And that comes with the Rogue Wave version of the Standard Library.

## The Rogue Wave Implementation

With a certain feeling of déjà vu, I embarked on an informal feasibility study. What would we need to do to convert from DinkumWare to Rogue Wave? The answer turned out to be removing the allocator template parameters, adding a few namespace directives and setting the include directories in the IDE. Not too painful.

I have only had two or three weeks to get to know the Rogue Wave implementation. The biggest problem so far has been that standard strings don't seem to work in the DLL version of Borland's Run-Time Library. In trying to track down the cause I spent far too many hours poring over the string code, both in the header files and in the assembler code in the CPU window of the debugger. In the end I gave up and switched back to the static Run-Time Library, which mysteriously cured the problem.

It was also necessary to be explicit about the library namespace, std. VC++ 4.0 did not support namespaces, so we had configured the ObjectSpace library not to use them. The DinkumWare implementation either didn't put the STL in a namespace or included a "using namepsace std;" directive (I forget which). The Borland/Rogue Wave package conforms to the standard; it puts all the standard declarations in namepsace std and does not provide a using directive. This meant adding "std::" to declarations of vectors, lists, etc. in header files or adding the appropriate using directive in .cpp files.

One more small code change was necessary (due to a change in the Standard, I think): the Rogue Wave erase() functions returned void, in the other implementations it returned an iterator to the next element. This presented no more than a minor inconvenience.

Rogue Wave's source code looks similar to that from ObjectSpace. It uses pre-processor macros to get round compiler limitations and is only moderately difficult to read. Both the ObjectSpace and Rogue Wave headers provide unsurprising implementations; in contrast, the DinkumWare code often chooses a less obvious technique.

## Conclusions

It would be a brave man who would pass judgement on STL implementations and I am not going to try. Rogue Wave have been writing class libraries for a long time and there is not doubt that they do a good job. P. J. Plauger is renowned as an expert on libraries, in both C and C++. ObjectSpace I know less about, but I very much like their product and their style.

All the STL implementations I have seen were written by better software engineers than myself. Even so, there are some conclusions that I think can be drawn.

First, and most important, I did not find any differences in behaviour. A vector in one implementation behaved in exactly the same way as a vector in the each of the others. Apart from the few things already mentioned I did not need to change the code. This is, of course, as it should be, but it is also an ideal that is rarely achieved. It is the precise definition of the STL that made this possible and I think we should congratulate Alex Stepanov and the C++ standardisation committee for this.

There is a world of difference between the original STL from HP and the complete C++ Standard Library. The HP implementation did not track the evolving C++ Standard and is

now out of date, but it is a perfectly serviceable package of generic containers and algorithms. The commercial STL vendors have kept up to date, but from a user's point of view this is a relatively minor advantage if all you want is basic containers and simple algorithms. The big difference is that DinkumWare and Rogue Wave supply the full Standard Library. ObjectSpace do not offer a complete C++ Standard Library as far as I know. They do, however, have a range of cross-platform libraries that provide facilities like threads that are not in the Standard Library.

So, if you need high-quality, cross-platform libraries with facilities beyond those of the C++ Standard Library, see what ObjectSpace have to offer. (Try http://www.objectspace.com.) If that is not important to you, I would suggest you use the Standard Library that comes with your compiler. And if you use Visual C++ 4.X I guess it would be worth upgrading to 5.0 on the assumption that some bug fixes will have been made in the Library.

I will offer one final piece of advice. The STL really is a quantum leap forward. Use it. Use it in preference to vendor-specific alternatives. Use it in preference to containers based on class hierarchies. And use the principle of generic programming on which the STL is based as another tool in your software developer's toolkit.

*Phil Bass*
*Phil@stoneymanor.demon.co.uk*

# Reviews

## Java 1.2 and JavaScript for C and C++ Programmers

**Authors**: Daconta, Saganich, Monk, Snyder

**Published by**: John Wiley & Sons Inc

**ISBN**: 0-471-18359-8

**Format**: Softback 822pp w/CD-ROM

**Price**: 39.95 UKP

**Supplied by**: John Wiley & Sons Inc

### Target Audience

This book is primarily aimed at C and C++ programmers looking to move into Java programming but also tries to cover JavaScript within the context of HTML programming and Java applets

### The CD

As with many other books, the CD contains all the source of the book but also adds many demos and trial versions of development tools, along with Voyager from ObjectSpace and their Java Generic Library (STL for Java).

### The Book

The style of the book is very chatty and easy to read although the introduction contains a lot of the standard hype about Java being the 'way forward'. I was rather disconcerted by the number of silly typos and poor grammar which a careful proof review should have caught - indicative of a book somewhat rushed to market.

A standard example is used throughout much of the book as a teaching aid, that of a bookshelf containing a series of books. Immediately, poor design rears its head with the book class inexplicably containing a 'next book' member used by the bookshelf to manage its list of books. The examples are a bit sloppy too with the program output not always quite matching the actual code shown. This will confuse beginners and, again,

should have been caught in the technical review of the book prior to publication.

The book goes on to compare C & Java, then C++ & Java, and this shows up some holes in the authors' knowledge of C (especially the ISO standard preprocessor) and C++ (especially the explanation of the scope resolution operator :: in relation to global names). The worst example of this begins on page 126 where RTTI in C++ is being compared to Java: the rather contrived C++ example has a virtual method which is overridden in two derived classes, yet still uses 'typeid' and a downcast to determine which method to actually call. The supposedly 'identical' Java code uses a direct virtual method call which is precisely what the C++ code ought to do, despite the mention of 'instanceof' which is the closest Java equivalent to RTTI!

Even the authors' knowledge of Java is called into question by comments such as "[the 'super' variable] is conceptually identical to the 'this' variable". When discussing inner, local and inline classes) introduced in Java 1.1, the book managed to totally confuse me and even several re-readings didn't help clear the issue up until I went back to Sun's own documentation.

Given that the book is an extremely recent publication (1998), I was annoyed on behalf of several compiler vendors when the book blithely claimed that "Naturally, no compiler vendor has yet implemented the entire C++ Standard Library as specified in the latest draft standard". Having dismissed many of C++'s language features as "complex", it then praises STL and Daconta says "It would not surprise me if Sun did not add generics just to have the STL". Aside from the double negative, this book comes with a CD that contains ObjectSpace's JGL: a Java version of the Standard Template Library!

Throughout the book I found the Java examples to be unidiomatic although the C++ style of many of the programs probably makes them easier for a C++ programmer to follow. However, each example is attributed to its particular author and you can easily spot the differences in style, with some of the authors clearly being more at ease with Java than the others. Overall, the writing styles seem to blend well and the book does not often give the impression of being knitted together from four separate contributions.

Once the language has been introduced in some breadth, the book turns to a detailed look at specific standard Java classes, with good coverage of all the Java exceptions and how they occur, and it is at this point the book improves dramatically. Instead of contrived code fragments, whole programs are given with items of interest clearly highlighted in bold print. I still got the impression of a run-through of language and library features rather than a 'how to' approach which would be more productive in my opinion, especially where the Abstract Windows Toolkit is concerned.

After the AWT, the book moves on to beans and applets and then looks at the current flavour of the month: CORBA. RMI, DCOM and CORBA are compared and then IDL is explained followed by complete programs, again with items of interest clearly labelled. A brief detour through 2D graphics, maths and RMI is followed by a good exposition of security in Java, including signed JAR files and cryptography, again with complete examples showing how each feature or class works.

At this point the book dips again with a very cursory look at 'java.sql' which, to my mind, fails to clearly explain the transactional elements of the code examples given. Perhaps it is unfair to expect a thorough grounding in relational database access at this point but some more 'hints & tips' would make the section more worthwhile.

Next up is internationalisation which is quite well explained, showing how to write an applet that uses locales to provide a multilingual interface - something that more programmers would do well to consider in this increasingly global market! Microsoft's ActiveX and COM technologies are touched on next - I shall not comment being rather a fan of portability - and then a relatively short

section on JFC, the Java Foundation Classes, which could no doubt justify a book in its own right.

Finally, on page 749, we come to JavaScript with a quick run-through of the differences between Java and JavaScript followed by some short examples on how to use the two languages together to simplify form validation and so on.

## Conclusion

My initial misgivings about this book dissolved as I got further into it. I still don't believe it's a good introductory book for programmers coming fresh to Java, but for those programmers with some Java already under their belt and an appreciation of idiomatic Java, this book is useful as a ready reference for many of Java's associated technologies without being dry. It covers an extremely broad range of material and, from Chapter Six onwards, is generally good value for money. I shall be returning to those later chapters again and again as I gain more real world experience with Java.

*Sean A Corfield*
*sean.corfield@issolutions.co.uk*

# Beyond ACCU... Patterns on the 'net

In the world of software today there is one topic which seems to be mentioned in every single design discussion -- patterns!

Software design patterns are descriptions of deisgns which, over the years, have been proven to be useful in a variety of situations. Most will be aware of Gang of Four book (it's been mentioned in this publication before) which helped bring patterns to prominence over the last few years. Exactly what consitutes a pattern I won't try to pin down here, but I will try to point you in the direction of some interesting resources on the web. However, even if you don't have web access, perhaps only email, there are still ways in which you can participate in the worldwide discussion of patterns.

So, you want to find out about patters on the web? Where do you start? The first time I wanted to see what was out there I went through my usual actions...go to Yahoo, type "patterns" in the search box and sit back to see all those sites come up. And yes, they did come up - sites on knitting patterns, china patterns and clothes patterns! Software design patterns? Few and far between. Fortunately, I happened on a very interesting site that led to many others...

The place I started was `http://hillside.net/patterns/patterns.html`. This site is increasingly becoming a good jumping off spot for obtaining more information on patterns. It includes links to online tutorials about patterns, downloadable papers from the academic community, a variety of presentations given at past conferences, notices of upcoming patterns-related conferences and many other valuable resources.

It also provides a comprehensive listing of mailing lists you can subscribe to in order to partake in patterns discussions. These mailing lists encompass topics such as business patterns, IPC, consurrency and distribution patterns and CORBA patterns to take a brief selection. Be warned - the traffic on these lists can be quite heavy and some of the discussions very abstract!

While monitoring some of the discussions on these mailing lists it is often possible to pick up on further web sites that have interesting material on them. One in particular is that of Brad Appleton, located at `http://www.enteract.com/~bradapp/`. This site contains one of the most complete resource lists I've seen in a while! His list of pattern related links (`http://www.enteract.com/~bradapp/links/sw-pats.html#Sw_Pats`) is one I frequently revisit. It gives information on tutorials, papers, further sites of interest to patterns and lists of "Patterns User Groups", where people get together to have discussions. However, you will find the last mainly restricted to the US!

Another site which is perhaps more generally relted to object oriented matters is Cetus at `http://www.cetus-links.org/`, which boasts "8757 Links on Object-Orientation". Again, this has its own patterns section, but also provides many jumping off points for topics such as software metrics, frameworks, reuse and testing. It has a superb section on the many object oriented languages, with about 360 links on C++ alone!

I have only mentioned three web sites in particular, but even those seem to open up further avenues of exploration exponentially. The few I mentioned seem to be well targetted and have a good signal-to-noise ratio, so I would recommend them highly.

*einar@rhuagh.demon.co.uk*
*einarnn@atlan-tech.com*

## Credits

Editor

*John Merrells*
*merrells@netscape.com*

*c/o Einar Nilsen-Nygaard*
*65 Beechlands Drive*
*Clarkston, GLASGOW, G76 7UX.*
*UK*

*P.O. Box 2336,*
*Sunnyvale, CA 94087-0336,*
*U.S.A.*

Readers

*Ray Hall*
*Ray@ashworth.demon.co.uk*

*Ian Bruntlett*
*ibruntlett@libris.co.uk*

*Einar Nilsen-Nygaard*
*EinarNN@atl.co.uk*
*einar@rhuagh.demon.co.uk*

Production Editor

*Alan Lenton*
*alan@ibgames.com*

Advertising

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

Membership and Subscription Enquiries

*David Hodge*
*davidhodge@compuserve.com*
*31 Egerton Road*
*Bexhill-on-Sea, East Sussex. TN39 3HJ*

## Copyrights and Trademarks



## Copy deadline

All articles intended for publication in *Overload 27* should be submitted to the editor by July 1st, and *Overload 28* by September 1st.

PAGE 30 IS THE 32nd PAGE.

IT IS RESERVED FOR THE BACK COVER