ISSN 1354-3172

Overload

Journal of the ACCU C++ Special Interest Group

Issue 22

October 1997

Contents

Contents 2
Editorial 3
Software Development in C++ 5
Allocation Stats By Kevlin Henney 5
An Introduction to the UML By Richard Blundell 8
The Draft International C++ Standard 12
extern "X" and namespaces by George Wendle 12
C++ Techniques 15
Make a date with C++ And so to const By Kevlin Henney 15
Premature Optimisation By Alan Griffiths 21
Whiteboard 24
Using Algorithms-Sorting by Francis Glassborow 24
Rational Values Implementation Part 1 by the Harpist 27
Some Opportunities to Increase STL Efficiency By Sergey Ignatchenko 30
editor << letters; 36
ACCU and the 'net 42
Credits 44
Copyrights and Trademarks 44
Copy deadline 44

Editorial

C++

Not much has happened in 'John's world of C++' over the past couple of months. Mostly because I've been in the midst of a trans-Atlantic relocation. My wife and I have moved house, job, and country - a slightly stressful experience - and one which doesn't leave much time for software introspection. Except that I feel like my life instance has been deconstructed and reconstructed.

Class Life {} Merrills;

Unfortunately, I was instantiated at my new company with an incorrect attribute. They misspelled my surname. Not such a big deal, you might think. But, there are a lot of machines there are out there that know exactly who I am, and what I'm up to. I've been battling these evil boxes for two weeks now, all the time only having half an existence. Ever seen the Richard E. Grant film 'How to get ahead in advertising?' Well, the transition has felt a bit like that. Just as I get one entry repaired, I discover my incorrect details have been replicated to yet another database. It's not just the sign on my cubicle. It's my electronic mailbox, my real world mail box, my pay cheques, the key-card system, even my phone thinks I'm this other bloke. So finally, I think I've fixed everything. But, having created the real 'Merrells', I have to kill off this 'Merrills' doppleganger. A HR form is involved! The employment of 'Merrills' must be terminated because of incompetence, gross moral turpitude, or death.

Better Living Through Directories

What's the solution to all this replicated redundant people information? Well, every network should have a single store of network users, and their attribute values. The system administrator would only need to maintain a single database, and each of the evil network services and databases can then be directed to the central store.

'Directory Servers' do indeed exist, but are generally proprietary. NDS appears to be what's kept Novell alive for the past few years, and now they're pushing it onto other network platforms. NT's inability to scale up to large networks is due to the lack of a cohesive directory, but this should all be solved by the Active-Directory in NT 5.0.

Shouldn't there be a standard for this sort of thing? Well, X.500 is the ISO Standard for directory services, but it was designed by a committee, is very complex, and hence very expensive to implement. (Sound a little familiar?)

LDAP

In steps a nifty Internet directory access protocol standard. It defines how a client can access simple directory services like add, modify, search, and delete. Most system software companies are LDAP enabling their email client and server products at the moment. For messaging systems the directory simply provides another form of address book. Since the directory has a DNS address, and your machine is connected to the net, you can use any public directory server which supports LDAP. For instance there's a Internet white pages directory at http://www.four11.com

My new found Zeal

My LDAP fever is no coincidence. I've moved from Octel's Unified Messenger, based in London, to Netscape's LDAP Directory Server, based in Mountain View California. Since they advise, 'write about what you know best', I'll be filling some of these pages with my struggle to LDAP the world.

On to the more visceral experiences in life...

The Joy of Fry's

Time to introduce you to the mecca of nerd-dome. Picture this, the largest warehouse you've ever been in, decked out as a Mayan temple, filled with all goods required by the modern technophile. Yes! Resistors, cable ties, mother boards, processors, disks, PDAs, laptops, colour scanners, radio scanners, printers, books, CDs, DVDs, radar detectors, laser pointers, pagers, washing machines, walk-in fridges, and a TV for $9000. Oh yeah, Fry's started out as a supermarket, so they sell food too... Beef Jerky and Jolt Cola!

John Merrells
merrells@netscape.com

Copy deadline

All articles intended for publication in *Overload 23* should be submitted to the editor, by November 14th.

Software Development in C++

Allocation Stats
By Kevlin Henney

Last issue [1], John looked at counting object allocations as a way of detecting creation and destruction inconsistencies for objects of a particular class. This approach detects leaks at the object rather than the memory level, which is what overloading the `new` and `delete` operators will give you. It is also significantly simpler, more reliable, and allows you to focus on objects of a specific class than simple instrumenting of the global allocation operators would allow.

John closed the article with some thoughts you might like to ponder. Rather than pursue these here, I've decided to follow up the theme of basic stats collection he introduced and revisit raw memory allocation. Given that you can only control what you can measure, collecting data on memory allocation can lead to more than bug squashing; it can also give you some idea of the behaviour of your system in terms of its resource use.

Defining the interface

A collection of statistics is an identifiable concept we can capture as a class. The `new` and `delete` operators will then manipulate an instance to log the statistics, and a direct call or indirect callback may be used to retrieve and report them. This leads to the following class definition:

```
class alloc_stats
{
public: // counter type

    typedef unsigned long count_type;

public: // construction

    alloc_stats();

public: // logging

    void log_allocation(
                   size_t sizeof_alloc);
    void log_deallocation();

public: // reporting

    count_type allocated() const;
    count_type deallocated() const;

    bool       balanced() const;
    count_type over_allocated() const;
    count_type over_deallocated() const;

    size_t min() const;
    size_t max() const;

    double mean() const;
    double variance() const;

private: // state

    count_type alloc_count;
    count_type dealloc_count;
    size_t     min_size, max_size;
    double     total, total_squares;
};
```

Note the separation of principle interfaces: one is for logging, and the other is for reporting. If I wanted a more generalised design I would factor these out as interface classes – abstract base classes containing only pure `virtual` functions – and mix them into `alloc_stats`. However, as described here it is a simple concrete class without any polymorphism.

I have only collected a few stats. Note that all of the stats collected are low overhead as they do not require long lists of remembered info:

1. Allocation and deallocation count, which will hopefully be the same;

2. Smallest and largest object allocated;

3. The total number of bytes allocated, which will give the mean size for an object, and the total of the squares of object size, which can be used to calculate variance and in turn the standard deviation.

The numeric types used are worth a mention. The type that expresses the size of an allocated object is `size_t`, and so it is not surprising to see it used for `min_size` and `max_size`. In a long running system it is likely that the accumulated totals would

overflow an integer, and so a `double` has been used for its range. The canonical counter type is `unsigned long`. Or rather used to be. For one reason or another on many systems it may not be the widest the integer type; some systems will have `unsigned long long`. As the counter type should be the widest type, and as this may change, the design decision has been captured as the `count_type` and the interface written in terms of it.

Defining the implementation

The construction is a fairly straight forward zero initialisation:

```
alloc_stats::alloc_stats()
  : alloc_count(0),
  dealloc_count(0),
  min_size(0),
  max_size(0),
  total(0),
  total_squares(0)
{
}
```

The logging functions are not too surprising. However, threadsafe they are not. Some kind of lock (e.g. mutex) is required to ensure that `log_allocation` works safely. The same lock could be used for `log_deallocation`, although an interlocked increment will perform the same job if you have one available for the `count_type`.

```
void alloc_stats::log_allocation
                   (size_t sizeof_alloc)
{
 ++alloc_count;

 min_size =
       std::min(min_size, sizeof_alloc);
 max_size =
       std::max(max_size, sizeof_alloc);

 total         += sizeof_alloc;
 total_squares += sizeof_alloc *
                            sizeof_alloc;
}

void alloc_stats::log_deallocation()
{
  ++dealloc_count;
}
```

The reporting functions are simple queries, and are potential candidates for inlining. If you don't have a stats book to hand, the most useful feature of the code below is the calculation for variance.

```
alloc_stats::count_type
alloc_stats::allocated() const
{
    return alloc_count;
}

alloc_stats::count_type
alloc_stats::deallocated() const
{
  return dealloc_count;
}

bool alloc_stats::balanced() const
{
  return alloc_count == dealloc_count;
}

alloc_stats::count_type
alloc_stats::over_allocated() const
{
  return alloc_count > dealloc_count
         ? alloc_count - dealloc_count
         : 0;
}

alloc_stats::count_type
alloc_stats::over_deallocated() const
{
  return dealloc_count > alloc_count
         ? dealloc_count - alloc_count
         : 0;
}

size_t alloc_stats::min() const
{
    return min_size;
}

size_t alloc_stats::max() const
{
    return max_size;
}

double alloc_stats::mean() const
{
    return total / alloc_count;
}

double alloc_stats::variance() const
{
    return
      (total_squares / alloc_count)
       - (mean() * mean());
}
```

Replacing *new* and *delete*

We need an object to collect stats about our object allocation. You can leave the following as a global, make it a file scope `static` or place it in an anonymous namespace, according to taste and intent:

```
alloc_stats objects;
```

As the aim is to instrument rather than redefine the basic behaviour of allocation, you should follow the convention of the

existing operators [2], taking into account the latest spec for these operators [3]:

```
void *operator new(size_t size)
{
  // cannot allocate null
  // pointer for size == 0
  //
  void *ptr = malloc(std::max(size, 1));

  while(!ptr)
  {
    // get current new handler
    void (*handler)() =
                     set_new_handler(0);
    set_new_handler(handler);

    if(handler)
    {
      handler();
    }
    else
    {
      // sadness and woe :-(
      throw bad_alloc();
    }
  }

  // only log allocations
  // when successful
  objects.log_allocation(size);

  return ptr;
}

void operator delete(void *ptr)
{
  // do not count delete on
  // a null as a deallocation
  if(ptr)
  {
    objects.log_deallocation();
    free(ptr);
  }
}
```

Reporting back

Collected stats are of no use unless they are reported. As we have expressed our design with a class, it seems natural to use `operator<<` for output. For brevity I have kept this output simple, but you could report on the standard deviation by taking the square root of the variance, or you could add an explicit field for reporting the difference between the number of allocations and deallocations [1]:

```
ostream &operator<<(ostream &out, const
alloc_stats &src)
{
  out << "objects allocated:   "
      << src.allocated() << endl
      << "objects deallocated: "
      << src.deallocated() << endl
      << "smallest object:     "
      << src.min() << endl
      << "largest object:      "
      << src.max() << endl
      << "mean object size:    "
      << src.mean() << endl;
  return out;
}
```

[1] Note that this is not necessarily the same as the number of leaks, as there may be memory trampling errors, i.e. a single object deallocated twice and two objects forgotten is not a leak of one object.

When to report these? You could report them at the end of `main`, or at some point after. I won't get drawn into how to try and make sure that all of the destructors for static storage objects have been called, but I will say that if you really do want it last thing (or as near as possible), try to avoid using I/O stream objects as they will probably have been destroyed. As a design note, it is generally inadvisable to have too many static storage objects with sophisticated allocation behaviour.

For exposition I have kept it fairly simple:

```
void report_objects()
{
  cerr << objects;
}

int main()
{
  atexit(report_objects);
  ...
}
```

Closing thoughts

The mechanism shown collects stats for all object allocations, so...

- What other statistics do you feel might be useful to collect? How difficult would it be to add these?
- How would you extend the code to also cover array allocation?
- How could you recast the code to handle stats collection on a per class basis? The solution should be relatively non-intrusive, i.e. only a minor modification to the definition of the class of interest need be modified to add the facility. The solution to this will give you an answer to one of John's questions: factor the feature out as a mix-in class [Hint: it

should be an empty class, and templates play a part in the action].

- Given that you have per-class stats collection, how would you reuse the same code to implement a global stats collector again?

References

1. John Merrells, "Object Counting", *Overload* 21.
2. Scott Meyers, *Effective C++*, 1992, Addison-Wesley.
3. *Working Paper for Draft Proposed International Standard for Information Systems – Programming Language C++*, `http://www.maths.warwick.ac.uk/c++/pub`.

Kevlin Henney
kevlin@acm.org

An Introduction to the UML
By Richard Blundell

Until relatively recently, the Unified Modelling Language (UML), the Object Modelling Technique (OMT), Object-Oriented Software Engineering (OOSE), Booch, etc., were terms I heard bandied around a lot, but I only had a vague idea what they actually involved. I was familiar with less formal methods of describing software systems. If I needed to describe a class, a class hierarchy or a collection of interacting classes to a third party, I could improvise a diagram containing various boxes and lines, and then (if I got confused looks all round!) explain to what the particular elements in my diagram referred. I was interested in finding out about more formal or standardised ways of conveying this information. Such methods would also be useful for documenting the architecture of a software system (which is, of course, the same thing as describing it to someone else, only you don't know to whom you are describing it in advance…) I did not, however, have access to much information about these formal methods, and I got the opinion that unless I worked on a large project for a large company that had a heavyweight development policy, I would not be able to learn or use such methods.

Deep down, however, I felt that such standard methods were no different in principal from my informal sketches. In other words, there was nothing intrinsically difficult about such methods, even though such diagrams were often confusing at first sight. After investigating the UML, I now know that my preconceptions were substantially correct.

I have always believed that with programming languages, once you have learnt one, and have gained a certain amount of experience using it, then switching to any of the other mainstream languages is largely a matter of learning the syntax of the new language, and then spending a variable amount of time learning its unique idioms and pitfalls. This latter stage *can* take a large of time, but I believed that a competent programmer could be productive soon after learning the new syntax. (Some language shifts are harder than others, e.g. non-OO to OO, but in some ways the new programming paradigm or model primarily involves a new set of idioms, albeit a big one.) It is the same with modelling languages. If you already know one, switching to the UML should be fairly easy (in fact, this was one of the design criteria). As long as you know the architecture of what you are trying to describe, and understand how the classes and objects within it interact, it should be moderately easy to start using a modelling language by learning the basic syntax.

Intentions

I hope that what follows will help to describe and explain the notation used in the UML, some of the benefits of using it, and allow you to document systems and communicate your ideas. In this article I shall cover, as background, a brief history of the UML. I

shall then describe the modelling elements required for simple static class diagrams, because I feel that these will be most useful to those not currently employing a modelling language, as well as to those programming more as a hobby. These diagrams allow single classes, class hierarchies and related classes to be described, and are a good introduction to the UML before burrowing deeper. I will naturally leave a number of things uncovered, but I hope to describe these and most of the remaining capabilities of the UML in subsequent articles. On occasion I shall highlight differences between UML notation and OMT/Booch.

In what follows, I shall assume familiarity with the concepts of classes, member variables (attributes), methods (operations) and inheritance, and basic C++ syntax (The reviewer of this article wanted to stress that UML is programming language independent. My discussions will be mainly for this C++ audience, however).

History of the UML

Modelling languages have been around for a long while, and when object-oriented development methodologies were first introduced twenty-odd years ago existing modelling techniques were modified to support this programming paradigm. Many different languages were invented, but by the early 1990s, a few leading techniques were in mainstream use. These included OMT, Booch and others, and they helped their practitioners communicate ideas and specify and document their systems. The diversity of techniques produced a desire to create a unified language with which all reasonable systems could be described without ambiguity. In 1994, Grady Booch and Jim Rumbaugh (OMT) got together to create the Unified Method, combining the best aspects of their two methods. In 1995, Ivar Jacobson joined the team and merged his OOSE method with the Unified Method to create the first specification of the UML.

After much public feedback, version 1.0 of the UML was published in January 1997. As a result of subsequent feedback and experience from the public and UML partners, version 1.1 of the specification is due for release in September 1997.

Very little of the specification is new and unique to the UML. The UML largely represents the bringing together of the best elements from all the current modelling languages, thereby building on the individual strengths of what went before. Some new elements have been added, however, including support for threads, concurrency and distribution, components, and direct support for patterns.

The UML has been designed to be able to describe all phases of the software development process (as well as non-software oriented processes - the UML definition is itself presented in UML notation!). The UML is a specification of a meta-model - a language that can describe models in general, and the most visible aspect of the UML is in the numerous types of diagrams that its notation supports. It has also been designed to be extensible, and can therefore be used to describe most systems that were not envisaged or encompassed by the base definition.

The static class diagrams that I describe below are just one of the types of diagram defined. Other types of diagram allow descriptions of classes and objects, interactions between them, their allowed state transitions and lifetimes, class and file dependencies, right through to packaging details and implementation details.

UML has support for *Use-Case* modelling, which aids requirements analysis, both static and dynamic modelling of classes and objects, *Component* modelling, which helps to organise classes and functionality into pre- and post-compilation files to aid re-use, and *Deployment* modelling to specify how your system is partitioned and distributed across a network.

Class Diagrams in the UML

OK, let's get stuck in. A *class* can most simply be shown as a rectangle with the name of the class in it. This is similar to the OMT class notation (as are most of the diagrams in what follows), but quite different from Booch, which shows the class name in a sort of splat shape with a dashed outline. A UML example hedgehog class is shown in figure 1. (I apologise if the lines in the figures come out rather faint.)

Fig. 1 - The hedgehog class with details suppressed.

One feature of the UML is its acknowledgement of design tools, and so the ability to show different views of a particular system is supported. Certain features can be suppressed or displayed at will, depending upon the level of detail required. The example in figure 1 shows a class with all external interfaces suppressed, leaving just the class name.

A less-suppressed model of the same class is shown in figure 2. Here we see that a class is shown as a box divided into three compartments. The top *name compartment* holds the class name, the middle *list compartment* holds the class attributes, and the lower list compartment holds class operations.

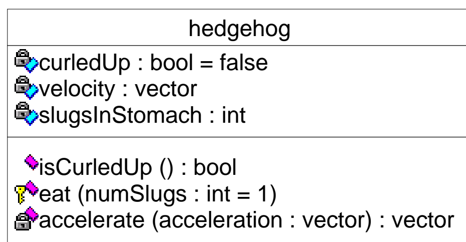


Fig. 2 - More detail of the class.

Class Attributes

The general format for an *attribute* is:

```
[vis] name : type [= init]
```

where `vis` is an optional visibility symbol, `name` is the attribute name, `type` is its type, and `init` is an optional initial value. Visibility symbols defined in the UML include +, # and – for public, protected and private respectively. In addition, the "+$" visibility denotes a public class (i.e. static) method. However, many UML tools, such as the one I used to create the figures, choose to use icons for these attributes instead (a freedom endorsed by the UML), and so you often don't see these symbols in use. An extra visibility is supported - that of implementation, which means that the item is visible only within the current *package*. I shall defer the definition of a package until later. An ellipsis at the end of the attributes list indicates that further attributes exist but are not shown.

Class Operations

The format for an operation is:

```
[vis] name (parms) [: type]
```

Here, `vis` is the optional visibility as before, `name` is the operation name, and `type` is an optional return type, assumed to be void if absent. `parms` is a comma-separated list of parameters to the function, each one taking the form:

```
name : type [= default]
```

where `default` is the default value for the parameter if it is not specified.

Class Associations

Associations between classes are indicated using a solid line joining the two class symbols. An example of an association is shown in figure 3. Association lines can be labelled in a number of ways to describe the meaning of the association. *Role names* can be added at each end of the line to describe each class' participation in the association (figure 4).

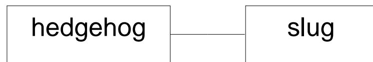


Fig. 3 - A binary association.

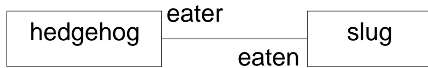


Fig. 4 - Association roles.

The *multiplicity* of each end of an association can be added as well, to indicate how many classes are involved in the association (this differs from OMT and Boock - see figure 5). Common formats for multiplicity include "n", where n is a number, or "n..m", indicating a range of values between n and m. An asterisk is used to indicate a potentially unlimited number, and a single asterisk is a shorthand for "0..*", meaning any non-negative number.

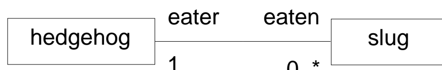


Fig. 5 - Roles and multiplicity.

Aggregation

If one class forms some form of *aggregation* of other classes - a box holding a number of paper clips for example - then this can be shown using the aggregation symbol, a hollow diamond, on the aggregate end of the association, as in figure 6. An aggregation often denotes that the aggregate holds references to all of the things that it is aggregating, but is otherwise independent of them. It therefore indicates no ownership.

Fig. 6 - An aggregation association showing the multiplicity of the aggregation.

Composition

A stronger form of aggregation is *composition*. In composition, the lifetime of the aggregater and the aggregatee are similar (although the aggregate can be built dynamically, or elements can be removed before the aggregate is destroyed) so that when the composite is destroyed, the constituent parts are destroyed with it. It therefore indicates some form of ownership. Figure 7 shows an example of a composite, the composite association being indicated by a filled aggregation diamond. Composition can be shown in a number of ways, two of which are shown. The other two that I know of involve showing the aggregated class symbols within a larger class symbol for the aggregate, but I don't know how to get my modelling tool to produce those forms! A class is a composite if it contains the aggregated classes by value, although I do not believe that this need be the case.

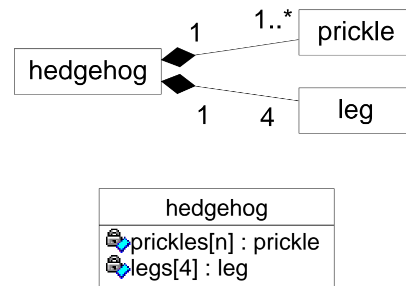


Fig. 7 - Two ways of showing composition.

Generalisation and Inheritance

Class inheritance is shown using a *generalisation* arrow leading from the more derived class to the more general one, as shown in figure 8 (OMT users will notice that the generalisation symbol is not quite the same in UML). Multiple inheritance is shown using more than one generalisation leading from the derived class to its parent classes.

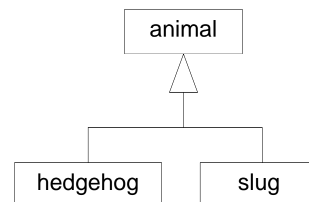


Fig. 8 - Generalisation.

As a final example, a slightly more substantial diagram is shown below in figure 9, which combines many of the ideas covered so far. This example also illustrates notes, the rectangle with the folded top corner, used to annotate a diagram. It also illustrates two alternative ways of showing generalisation, with either combined or separate arrow heads. If what I have been describing above made any sense, the figure should stand without any further explanation!

Conclusion

There is a lot more to the UML than I have covered here - many more concepts, many more diagram types, and many more refinements to the concepts I have illustrated above. What I have covered this time, however, should be sufficient to allow you to start using the UML to share information about systems you are developing in a standard fashion that is rapidly growing in popularity. Next time I hope to cover collaboration diagrams for documenting design patterns, amongst other things. As usual, feedback is welcome.

References

Rational web site is at www.rational.com, which contains a lot of UML information.

Richard Blundell
rpb@mail.ndirect.co.uk

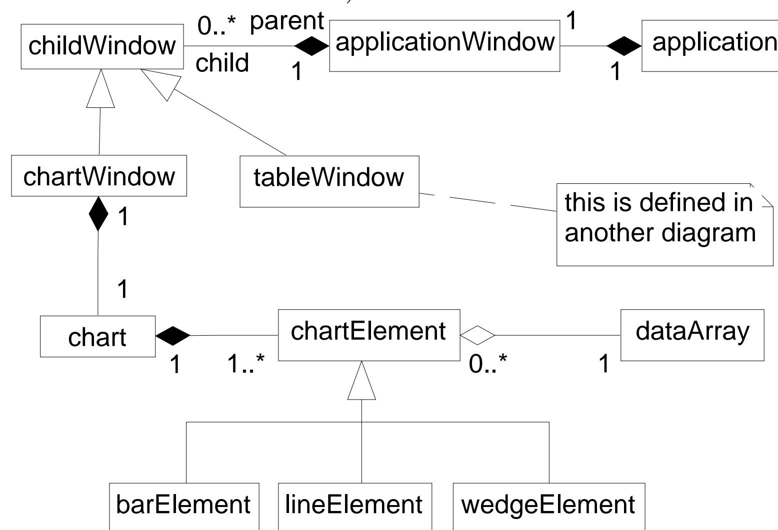


Fig. 9 - An extended example, showing a breakdown of a charting application.

The Draft International C++ Standard

extern "X" and namespaces by George Wendle

One of the features that highlights the feeling that C++ is settling to an agreed standard is the way that many of the compilers are converging. Most compilers either already support such things as namespaces, or plan to do so with their next release. The worst case scenario is that SC22 decides that the actual document presented as a Committee Draft has had too many changes made to it in order to a reach consensus of the participating National Bodies (like the UK's BSI or what was ANSI in the USA and I now understand is NITS) to allow it forward as a Draft International Standard. Even in this case I believe that there is now a general feeling that we all agree on about what constitutes 'standard' C++.

The spectre that I see hanging over us is that some major compiler implementors may feel that they are bigger than the standard and abrogate privileges to themselves that even X3J16/WG21 has refused to themselves. X3J16/WG21 has placed all of the Standard C++ Library into a `namespace` with the solitary exception of those items inherited from the Standard C Library. Even in the latter case it has chosen wording to strongly encourage implementors to wrap that part in `namespace std` as well.

Superficially it would seem that companies such as Borland and Microsoft should instantly wrap their proprietary libraries (OWL, MFC etc.) in suitable `namespaces`. Their failure to do so already ensures that name clashes are well nigh inevitable. Why haven't they? I can only guess. I suspect that there maybe a backward compatibility problem with DLLs and link-names. I do not know enough about the technology to do more than speculate. However this is a matter that needs urgent attention. Pure, single language environments such as Java-JVM can solve linkage problems because the combination has complete control of the execution environment. Any problems between JVM and the underlying platform are fixed up by some mechanism that is completely invisible to the user (well not quite, because you often have to set up critical features such as home directories before you enter JVM.) However what we want is that the components, be they ActiveX, JavaBeans, VB controls or whatever, have interfaces that completely hide the implementation details. I should not care what language a DLL has been written in. Perhaps this is several steps further than the tool providers are ready to go.

We must distinguish between pure C++ resources and those provided by some external mechanism. Items such as OWL/MFC are described as C++ libraries. These should be provided in the correct C++ form; that is they should be encapsulated in a vendor specific `namespace`. Any mechanisms needed by other languages (Visual Basic, Delphi etc.) are problems for those languages and should be fixed by them. I can think of many ways of doing such things but I leave it to those with expertise in component interfacing to address those and perhaps explain to the rest of us what such things as SOM, COM etc. do to help/hinder.

C++ already provides the mechanism for interfacing (linking) to external 'components'. The problem is that so many view the mechanism in a limited historical perspective. Most C++ programmers know that they sometimes have to declare a function as being `extern "C"` so that its linkage name will be compatible with a third party C library (comms., network, graphics etc.). What most do not realise is that there is much more to that facility. Basically when I declare a function to be `extern "X"` I am specifying something about the function interface. Note that I am not saying anything about the implementation. Let me clear that up before I go on.

```
extern "C"
{
  void fn(mytype const * i)
  {
    cout << *i;
  }
}
```

Is perfectly alright. The use of C++ in the body of the function is perfectly acceptable. The `extern "C"` simply provides information about how the function may be called. The first and most obvious feature is that it requires that the internal source code name `'fn'` be provided externally in whatever form this implementation uses to link to C code (usually without name-mangling but it might be otherwise if the linkage was to object code produced by a C compiler that used some form of name-mangling to support type-safe linkage (for example the old JPI TopSpeed C compiler). The consequence of this is that C code can probably call that function though this is not something that C++ can legislate for.

There is a second subtler requirement that caused considerable thought among those developing C++; how are arguments to be passed? Without going into detail, there are several ways that arguments can be passed from the point of call to the point of execution of a function. For example if a function takes two arguments and they are going to be passed on a machine stack, which order should they be placed in? C and Pascal have opposite default arrangements. C++ basically says that the order is generally of no concern to the programmer. That last statement is generally true until you are linking to external code that has not been provided by the same compiler.

Now there is nothing to prevent Microsoft from implementing `extern "Visual Basic"`, or Borland from implementing extern `"Microsoft C++"`. The former would enable the simplest possible reuse of a VB function from C++ code while the latter could be used to ensure that the linkage details of a function call were compatible with object code compiled with a Visual C++ compiler.

What I am trying to get at is that C++ already has mechanisms built in to it to allow management of mixed programming from the C++ side. It even has a mechanism that provides some support for exporting its functions for use by modules written in other languages. Of course using `extern "X"` can result in problems when porting code but at least these problems will be clearly identified. Fixing it may not be in the programmer's domain, though sometimes it will be. For example porting source developed with Borland C++ to Visual C++ might simply mean that you had to suppress the `extern "Microsoft C++"` declarations.

As a programmer I am powerless unless my compiler provider includes supportive `extern "language spec"` to help me. This is a quality of implementation issue. Start asking for it. Initially the response will be that no one wants it. If you give up it will be a self fulfilling claim. Perhaps we need to find a young, hungry tool developer who will understand that making their tools work with other people's object code will provide essential extra value. I am not going to embarrass anyone by naming names but I can think of a couple of products with lots of potential and a small corner of the market who could benefit from such an approach. If I can simply drop in a new set of development tools without having to get new versions (if even possible) of all the libraries (Zinc, XVT, etc.) I am using I am much more likely to make the move. I will still have work to do on my own code but at least I will have the tools and support I need to do it.

Now let me turn to internal conflicts. I need those vendor provided libraries tucked into their own `namespaces`. I need to control what a name means. As C++ programmers mature they will stop writing:

```
#include <iostream>
using namespace std;
```

and replace it with something like:

```
#include <my_io_idents.h>
```

Inspection of `my_io_idents.h` will reveal `#include <iostream>` followed by a number of specific `using` declarations that only inject into global `namespace` those identifiers that I regularly use. Any others that I want can either be declared specifically or used via their elaborated name. This has two consequences; I can control the names in global namespace and I can document where less common names are coming from. For example suppose that two third party libraries both contain a `class complex`. I cannot use either of those libraries in code that contains `using namespace std;` unless the library has been encapsulated in a `namespace`. However if the library writers have had the courtesy to use the `namespace` facility then I can mix and match. Even if I elect to write `using std::complex` I can still use the

others by specifically qualifying `complex` with the relevant `namespace` at the point of use. Even if I lazily write something like:

```
using namespace std;
using namespace vendor_abc;
using namespace vendor_xyz;
```

I can still manage the problem of writing:

```
// results in an ambiguity error
complex x;
```

by writing

```
std::complex x;
```

Those that do not want to be bothered with what they consider to be unnecessary complications generated by `namespaces` can fix the problem with a `using namespace` directive. Those of us who are ready to move on will be able to do so. I want the freedom to choose. Don't you want the same? So which compiler implementor is willing to do the work so that I can use MFC encapsulated in a `namespace`?

All that is need for bad coding practices to proliferate is the silent acquiescence of good programmers.

George Wendle

C++ Techniques

Make a date with C++
And so to const
By Kevlin Henney

Summarising the previous article (*Overload* 21), C++ allows you to declare variables just about anywhere, and with a great deal of freedom in how you initialise them. There comes a point in every series of articles where `const` must be discussed and introduced, and this article will be the first place I do this. In theory there is not much to say, but in practice there is: standard C has `const`, which it in fact borrowed from C++, but there are still many programmers who do not use or understand it. As there are subtle differences between C and C++ in this area I will introduce `const` from scratch.

The essential motivation for `const` was stated somewhat ahead of its time (and I confess more than a little out of context) in the seventeenth century by Lucius Cary, Viscount Falkland:

When it is not necessary to change, it is necessary not to change.

Banishing magic

Magic numbers are the bane of any program. You are reading through some source code and you come across a calculation or limit check that involves one or more literals, and not a comment in sight. So what exactly does `509` mean in this context? Why did the programmer choose `17` as opposed to, say, `16`? `42` may be the answer to some things, but on its own in a piece of source code it tends to beg questions.

The good programming solution is to name the concept being represented. This applies right across the board: if you identify and understand a concept, name it and describe it. The name may be a sufficient description, but so long as there is something that plays the role of description you are creating a useful abstraction. This technique applies to types, functions and values. As well as providing a richer description, a named concept provides a single point of change in the event that the details of that concept are changed, e.g. in the case of a default file name.

What we want here is a named constant. The classic C approach (K&R as opposed to ISO) is to use the preprocessor:

```
#define DEFAULT_DIARY_NAME "diary.dat"
```

I have no love of the preprocessor. As a tool it is neither good nor bad, but in its application it often causes more problems than it solves. It is one of the greatest handicaps that C programmers take with them when they learn C++. In this respect teaching C++ to non-C programmers is easier as their understanding of programming concepts has not been damaged by use of the preprocessor.

The better solution, available in both C and C++ for this example, is to use `const`:

```
const char
default_diary_name[] = "diary.dat";
```

The advantage of `const` is that it is properly within the core language: it declares a named variable that is addressable, is visible in the debugger, and obeys the scope rules of the language in exactly the same way that a macro doesn't. Notice also that we no longer have any need to shout: upper case naming is a convention intended to prevent macros – which work by textual substitution without regard for scope and context – from trampling over other identifiers in your code. Many C++ programmers are still stuck to their C ways in equating macro naming with constant naming [2]. It may be a hard habit to break – and perhaps not even one you've ever questioned – but it is certainly more consistent, and ultimately worthwhile.

The effect of `const` is to allow queries but prevent assignment:

```
cout
  << "filename: "
  << default_diary_name << endl; // OK

// illegal: compilation error
default_diary_name[6] = 'f';
```

diff C C++

An interesting, but minor, difference between C and C++ is shown by the following fragment:

```
// legal C, illegal C++
const date unassignable;
```

That's right, C allows you to declare a `const` variable [3] – which is unassignable by definition – without initialising it! I'm not sure what was intended by leaving this hole in the language. No matter, C++ fixes it by doing the right thing.

In C all `const` variables are runtime constants, i.e. their values exist only at runtime and are not available to the compiler. The fact that one can have runtime constants in C and C++ is one of the strengths over languages like standard Pascal, where constants may only be simple compile time values (not even expressions). This allows the programmer to factor out important calculation based constants that cannot be determined until runtime:

```
date today();
int year_difference
              (date first, date second);

date dob;
cin >> dob.day >> dob.month >> dob.year;
const int age =
          year_difference(today(), dob);

// age may be used as a constant
// but may not be modified
```

However, there are a number of cases where a compile time constant is required. As the

---

[2] It is worth noting that the idioms for Java missed this difference and enshrined the upper case naming convention for constants. However, that is the idiom to be followed in Java. For C and C++ macros should use upper case and all other identifiers should conform to a consistent local convention.

[3] Yes, "`const` variable" is a strange term, but nonetheless a valid one in the context of programming. A programming constant is not the same as a true universal constant. That said, some theories suggest that not all universal constants are.

term suggests, a compile time constant is one that is known to the compiler and can therefore be substituted directly at compile time. In both languages there are places where a compile time constant is required:

- The size of arrays;
- The `case` label of a `switch` statement;
- The width of bit fields.

C also additionally requires compile time constants for aggregate initialisers and non-`auto` initialisers (this constraint was removed in C++, as described in the previous article). Note that all of these require integer types, so there is effectively no need for compile time constants of other types. In C you would not be able to use `const` to name compile time constants, probably leading to macro use:

```
#define MAX_DAYS_IN_YEAR 366
```

Or, more elegantly, an anonymous `enum` constant:

```
enum { max_days_in_year = 366 };
```

In C++ a `const` variable that has a compile time constant initialiser which is visible at the point of use is itself a compile time constant. This applies to both global and local `const` variables:

```
const int max_days_in_year = 366;
```

The compiler will substitute `366` wherever `max_days_in_year` is used, and will typically optimise away the storage for the variable – this won't happen if its address is used, and is unlikely to happen if you compile your code with debug options.

diff C C++ | more

There are some subtle differences between the linkage of a `const` at file scope in C and in C++: in C a global `const` has external linkage, i.e. `const` does not modify its linkage, whereas in C++ it has internal linkage, i.e. applying `const` has the silent effect of also applying `static`. This is a bit of a wart in the language, but I understand the motivation. The rationale is that programmers would (and should) use `const` where they might otherwise have used macros. As constants are normally bundled in header files, there might be a problem for some programmers if `const` had external linkage, i.e. they would probably be tempted to write:

```
// date.hpp
const int secs_per_day = 24 * 60 * 60;

// firstuse.cpp
#include "date.hpp"

// seconduse.cpp
#include "date.hpp"
```

This would cause multiple definition problems at link time. So the language bent towards the habit of the programmer, rather than the other way around which can be expressed as follows:

```
// date.hpp
static const int
secs_per_day = 24 * 60 * 60;
```

As a note, if you are writing header files that will be common to both C and C++ this is something you should note. If you want constants with external linkage, which will probably be most of them, you need to use `extern` in the definition:

```
// date.hpp
extern const date epoch;

// date.cpp
extern const date epoch = { 1, 1, 1970};
```

const and Pointers

Perhaps the greatest initial source of confusion with `const` relates to its effect on pointers. A pointer adds a level of indirection, and so there are two `const` aspects to consider: the pointer variable itself, and the object that the pointer references. Each of these aspects may be qualified `const`, but given syntax of pointer declarations the question is "how?".

```
const char *
default_diary_name = "diary.dat";
```

This is probably not all of what the programmer intended: it declares a pointer to `const char`, and not a `const` pointer. In other words, `default_diary_name` is assignable, even though its dereferenced contents are not:

```
// OK
default_diary_name = "/dev/null";

// illegal: compilation error
default_diary_name[6] = 'f';
```

To declare a `const` pointer, the `const` must be associated with the pointer variable name:

```
char *const
default_diary_name = "diary.dat";
```

```
// illegal: compilation error
default_diary_name = "/dev/null";

// OK
default_diary_name[6] = 'f';
```

Combining the two gives us what we intended:

```
const char *const
default_diary_name = "diary.dat";
```

```
// illegal: compilation error
default_diary_name = "/dev/null";

// illegal: compilation error
default_diary_name[6] = 'f';
```

How do we rationalise this? It is possible to dive into the syntax description of a declaration and rationalise it from the grammar upwards. However, there is a simpler approach for the most typical declarations [4]:

- What is pointed to lies to the left of the `*`, therefore `const char *p` declares a pointer to `const char`, which we know from before to be unassignable;
- What is doing the pointing, i.e. the pointer variable, is described to the right of the `*`, therefore `char *const p` declares a variable named `p` to be `const`.

You can arrive at the same conclusion by reading the declaration backwards: `const char *const p` declares p as a `const` pointer to `const char`. Note that the specifier ordering used here is the most common convention; an equivalent, but less common, declaration would be `char const *const p`.

With all that in mind, you should be able to make sense of the following:

```
const char *const day_name[] =
{
  "Sunday",
  "Monday",
  "Tuesday",
  "Wednesday",
  "Thursday",
  "Friday",
  "Saturday"
};
```

This is the `const` correct way of defining an unchangeable lookup table of immutable strings; many programmers tend to forget the second `const`.

Access rights

What role can pointers to `const` play? Do they, as the terminology suggests, point to `const` objects? Well, no, not exactly. This is one of those quirks of naming and history that you just have to live with. Bjarne Stroustrup's [*The Design and Evolution of C++*, Addison-Wesley, ISBN 0-201-54330-3] original idea was to have the qualifier named `readonly` by analogy with concepts found

---

[4] It is important to emphasise "most typical". As you may know, the language syntax allows far more complex declarations which you have to peel apart like an onion – and like onions, some complex declarations have been known to reduce grown programmers to tears (e.g. `signal`). However, if you are comfortable enough doing that, you probably don't need too many extra rules of thumb when it comes to `const`.

in operating systems, but followed advice to change it to const. This is a shame, as readonly would have better captured the intent.

Thinking in terms of *readonly* it is easier to see that a pointer to const does not necessarily point to a const object. Instead it guarantees that the pointer may only be used for read access, i.e. if it is a non-const object it may be modified by another route, but via a const pointer you may only observe changes not cause them. However, as a const object guarantees from the outset that it will not be modified, the compiler will reject any attempt to increase the permissions [5]. Therefore assigning a pointer to non-const the address of a const object is a compile time error. The following code fragment uses the day_name array defined previously:

```
// illegal: compilation error
char *illegal = day_name[0];
```

All this makes const a valuable specification tool. The type system offers a means to specify a system quite precisely. The types that you select, and the legal operations that you can perform on them, constrain the multitude of possible program behaviours to a subset of meaningful ones. With const you can now more completely specify the legal operations on your data. A function taking pointer arguments can easily show which objects are to be used for information only, i.e. const implies that they will remain unchanged, and which are to be operated on:

[5] I am assuming that you understand a cast will invalidate any such assumptions. I am unconcerned by this: it has always been understood that in the presence of casts most bets are off, including the rest of the type system's guarantees. Given that with a cast I can turn water into wine, it should be understood that all of the guarantees I give are for those co-operating with the type system – if you work against it, you are on your own.

```
struct diary;
void print(const diary *);
void update(
        diary *,
        date when,
        const char *description);
```

It is easy to understand the intent of these functions simply by inspecting the names and the const qualification (or absence of). To this end the const qualifier serves a definite purpose to the human reader, with the added bonus that it will be compiler enforceable. Before thinking about commenting your code, say as much as possible using the type system; the behaviour and assumptions left over are what you put in your comments.

I do not understand, or accept, programmers who say they like strong typing and then ignore const when it is offered and explained to them. It leaves the impression that they do not fully understand the relationship between the high level view they have of a system, i.e. its specification, and how best to represent it in code. In short, programming. Raising the semantic level of your programming by attaching meaningful behaviour to clumps of data is a fundamental theme I will return to in future.

Summary

- const is highly expressive and safe. The combination of inline functions and const makes many programming uses of the preprocessor redundant.
- There are subtle differences between C and C++'s interpretation of const: on the whole these are extensions, i.e. do not affect existing assumptions, but the difference in linkage of a file scope const is something to watch out for.
- It may be easier to translate const as "readonly", because const qualified values are not strict constants in the mathematical sense, and pointers to const may be point to non-const objects.

- `const` is a simple but powerful specification tool.

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

Premature Optimisation
By Alan Griffiths

Inlining

In Overload 21 Francis complained of programmers rejecting inlining, he then "justifies" the inlining of forwarding functions on the grounds that it has no space overhead.

As far as this goes this seems reasonable, however there are some significant hidden costs. For the sake of exposition let us assume that we inline all the "trivial" forwarding (and accessor functions?) in the class header files - separate "inline" headers don't affect the argument.

1. Firstly, the compiler needs to process and hold details of all these "free" inline functions for every class included in a compilation.

2. Secondly, if any of these functions ever changes then every file that depends on the header needs recompilation.

Both of these facts can seriously impact the build time of any serious software development by (1) increasing the resources required by a compilation and by (2) increasing the number of compilations. To give an example from "real life" I have seen this turn a one minute edit into a six hour rebuild!

Priorities

Looking at software development in more general terms we are always balancing cost of development (time, staff and tools), ease of maintenance, and runtime resources (space or time).

The correct balance varies from project to project, but I am sure that the project I'm currently working on is quite typical. In this, there are a few subsystems that are performance critical, but the vast majority of code must be delivered with an efficient use of development resources. (The usual comments about 80-20/90-10 rules are left as an exercise for the reader.)

Most code is not going to have a significant impact on the use of runtime resources and should be written to minimise the costs of development and maintenance. Simple, easily tested algorithms should be used, and various forms of code level "optimisation" (of which inlining is an example) should be postponed.

When to optimise

This approach leads to a functionally complete system (or subsystem) fairly early in the development life-cycle, but one that will fail in some cases to meet time or space constraints.

This is the time for measurement - it is amazing the unlikely places that the computer finds to spend its time or guzzle memory. For instance, last year I examined a test case that pulled large numbers of objects from a database system and presented them on screen. It was spending 30% of its runtime inside the string equality operator!

It was a matter of minutes to rewrite this function and only a couple of hours recompilation before I could confirm that I'd achieved a 27% speed improvement. (I HATE THE TEMPLATE INCLUSION MODEL! - Sorry, the speedup was worth the time spent, but with separate compilation I could have made the change in minutes.)

It is only when you can measure the use of memory or time within the application that it is possible to effectively control it. Otherwise how do you know what effect you expect from a change, or whether you have achieved it? (It helps justify the tools needed to achieve this that managers are also taught that "You cannot manage what you cannot measure.")

An additional benefit of only optimising when there is a functionally complete system is that the test harnesses and/or test scenarios should by then be debugged and stable. This allows changes to be validated quickly.

In practice, I've rarely found that the way an algorithm is coded has a big impact on the runtime, but I've frequently found that the choice of algorithm does have a big impact.

For example when there is enough RAM to hold "key/record number" pairs then "shell sort" or similar is an effective way to order records. However, when RAM runs out and virtual memory comes into play the performance of the same algorithm becomes quite poor. More sophisticated algorithms can be introduced to deal with large volumes of data but the trade-offs are often hard to predict and I would want to measure them to find the best approach. (For those that are interested I was reading about "polyphase merge sort" in the C/C++ Users Journal recently - the July issue I think - the exposition was clear, but I can't recommend the code that went with the article.)

Back to inlining

Returning to Francis's theme of inlining forwarding functions, we now know that there is a cost implied by this approach. We can also guess that it may reduce the code size or runtime marginally. Now, if we've measured the cost (either in code size or run-time) of keeping these functions "out of line" and found that we can't meet the system requirements without inlining these functions then we may be willing to pay this price. (Although, I would regard this particular technique as a last resort.)

A more serious issue with inlining is that it is too often used in a completely inappropriate manner (for example, to avoid creating a .cpp file and the extra typing concerned). I want to give an example of this, but all the cases I can locate raise additional issues which are beyond the scope of the current article.

A bad example

The following code is extracted from a current project (the author has already told me that he wishes he hadn't done it this way, so I don't think that he'll mind):

```
class MMapHier
. . .
// the MSI file for this hierarchy
//
string MSI() const
{
  return base->getName()+"HierMSI.PMH"
}
```

There follow forty similar functions including these added by a second programmer:

```
//BITMAP
// These functions shouldn't be here,
// should be in a cpp 'cos it's quicker
//
string bitmapGIDOffset() const
{
  return base->getName()+"HierBgo.PMH";
}

string bitmapChars() const
{
  return base->getName()+"HierBch.PMH";
}

string bitmapChains() const
{
  return base->getName()+"HierBol.PMH";
}

string bitmapExtent() const
{
  return base->getName()+"HierBex.PMH";
}

string bitmapMSIMap() const
{
  return base->getName()+"HierBmm.PMH";
}
```

This particular header is referenced throughout a large application. I don't know what the class is for - what it does is hold a base location in the file system and generate filenames for specific types of data. What is obvious is that every part of the system that relies on it for the generation of a filename needs to be recompiled whenever a new type of data is added.

Ignoring these issues and concentrating on the use of inline:

1. If any of the existing filenames change it will provoke a massive recompilation,

2. any compilation units that invoke these functions will have copies of the corresponding text constants,

3. these functions (which, for example, construct temporary string objects) are larger than is ideal for inlining,

4. the function call overhead is hardly a significant part of the cost of the function call, and

5. none of the runtime costs are significant compared to the cost of opening a file!

6. Every module that references this header devotes some compiler time to parsing it and some compiler memory to the bodies of these functions. For one header or one module this is not a lot but with a thousand or so "public" headers and a few tens of thousand modules these effects mount up dramatically.

Summary

Inlining is a code level optimisation technique that, like other optimisation techniques, can be abused. Like other optimisation it costs development resources. Optimisation is easy to do on a working (sub)system and hard to do on one that doesn't work. Adding complexity, of algorithm or of code, without demonstrable need is either recklessness or ignorance.

The fact that a method is a "trivial" forwarding function does not of itself justify inlining it.

Alan Griffiths
alan@octopull.demon.co.uk

Whiteboard

Using Algorithms-Sorting
by Francis Glassborow

Quite a bit has been said recently about the value of the STL containers. However these are only part of the STL. Just as important if not more so are the algorithms. Let me take a little of your time by considering the process of sorting in C++. The first thing to note is that STL algorithms provide specific performance characteristics. No longer do we have the uncertainty that accompanies use of qsort from the Standard C Library. We can also choose to from different sort algorithms depending on what we need. Fundamentally we have three sorts, each in two versions. If we are interested in a fast sort (but can tolerate a rare pathological case where performance deteriorates badly) we can use `sort`. If we need to preserve prior order (i.e. elements that compare equal will retain their existing order-not true of `qsort`) or we cannot risk the pathologically bad cases of `sort`, we can use `stable_sort`. Finally if I am only interested in the leading elements being correctly sorted (for example, if I just want the top three in correct order) I can use `partial_sort`.

Each of these three sort functions can be used with the default of smallest first (requires that `operator<` is meaningful for the items being sorted). Alternatively I can provide a comparison object (note that carefully, I will get to those shortly).

In the following code examples I am going to focus entirely on sorting. I know that other parts of the STL support other aspects of the following code but I want to keep a clear focus in this series of articles and leave it to you to merge all the aspects into a single whole. Consider:

```
void getValue(int * entry)
{
  cout << "Enter an integer:";
  cin >> *entry;
  // code to handle non-integer
  // entry note that quite a lot
  // of polish could be provided
  // in this function
}
```

```
int main()
{
  int array[100] = { 0 };   // zero an array of 100 ints
  int i;  // ensure i remains in scope
  cout << "Enter up to 100 integers, terminate with any value above 9999" << endl;
  for (i=0; i<100; i++)
  {
    getValue(array+i);
    if (array[i] > 9999) break;
  }
  // at this point i will point
  // one beyond last valid entry
  switch (i)
  {
  case 0:
    cout << "No values provided";
    break;
  case 1:
    cout << "Single value " << array[0] << "Cannot be sorted.";
    break;
  default:
    sort(array, array+i);  // LINE X
    cout << "The sorted values, smallest first are:" << endl;
    for(int j=0; j<i; j++)
      cout << array[j] << endl;
  }
}
```

Note how easily the design works. STL algorithms expect an iterator (`int *` in this case) to the first item and one to just beyond the end. I have not had to track the results of an early finish. `i` will always reference the next potential `int`; either 100 if the whole array has been filled or the entry containing the termination value.

Now what happens if I want to sort in descending order? No problem as long as I remember a couple of simple details. I must `#include <functional>` to get the relevant templates and I must remember that I am dealing with function objects (later I will write an article specifically on these). Now I replace `LINE X` in the above with:

```
sort(array, array+i, greater<int>());
```

In other words I pass `operator()` for the instantiation of the template class `greater` for an `int`. Do not worry about the magic at this stage, just enjoy the ease of implementation.

Now suppose that I want to sort my values based on the tens digit only. In other words 21, 113, 47, 2 will finish as 2, 113, 21, 47. No problem. First I must create a suitable comparison object:

```
bool compareTensDigit(int i, int j)
{
  // discard units values
  i /= 10;  j /= 10;

  // retain original tens place
  i %= 10; j %= 10;

  return i < j;
}
```

Now I can replace `LINE X` with:

```
sort(array, array+i, compareTensDigit);
```

Now what would happen if I replaced `LINE X` with:

```
sort(array, array+i);
sort(array, array+i, compareTensDigit);
```

I might have intended that all items with the same tens digit would be ranked in numerical order, smallest first. If that happens I am just lucky because the algorithm used for `sort` is selected for speed not for preserving previous structure (in other words it has the characteristics of `qsort` except that there is a specified performance requirement). What I need to do is to write:

```
sort(array, array+i);
stable_sort(
          array, array+i,
          compareTensDigit);
```

It might be a little slower (immeasurably in the current example) but it will do what I want. I can use the fastest available algorithm first time round and then preserve existing structure by using calls to `stable_sort` thereafter.

Now suppose that I am only interested in the top three values from those typed in. I replace `LINE X` with:

```
partial_sort(
        array, array+3,
        array+i, greater<int>());
```

Note that this sort does not work very well if I want the top elements from a multiple sort because, as far as I know, `partial_sort` does not preserve prior structure. I think this makes sense as I would only use `partial_sort` when I was wanting the highest possible performance.

There is a last item that should be dealt with under the heading of sorting, that is the minimalist case where I just want to get an element right with everything before it unsorted but of an appropriate size and everything after it likewise. For example given 5, 7, 2, 3, 9, 1, 1, 7, 3, 4 and a pivot point at the fourth element the following would be fine: 1, 2, 1, 3, 7, 5, 3, 9, 7, 4.

Replacing `LINE X` with:

```
nth_element(array,
            array+(i>>1),
            array+i);
```

will make the first half of the array contain the small values and the second half the larger ones. The median value will be the

middle one (well almost if there are an even number of items). Like the other sort algorithms this can take an extra parameter to provide a comparison object (either a function address or a full function object)

Well I think that is enough for now. Please try to abandon your C programming techniques when writing C++ and move on to the high level tools provided by the C++ of the late 1990s.

Francis Glassborow
francis@robinton.co.uk

Rational Values Implementation Part 1 by the Harpist

Following on from my previous article discussing the design of a 'Rational Type' class, I shall consider some implementation ideas.

For the purposes of this column I am going to assume that `integer_type` (Francis suggests that the `_t` suffix is probably reserved to Posix, can anyone clarify that?) is implemented as `unsigned int` so the definition of `Rational` includes:

```
public:
  typedef unsigned int integer_type;

private:
  integer_type numerator, denominator;
  mutable long double fp_value;
  bool negative;
  mutable bool converted;
```

Now as this is a simple value oriented type I would like to leave the copy constructor and copy assignment to the compiler generated defaults. Under the rules the compiler is more likely to be able to generate maximum efficiency copies if I do not try to interfere (it can probably use a direct memory copy, at least until `integer_type` ceases being a built-in type.) However this requires that all memory for the object being copied has been initialised. So I will need to attend to the constructors in a little more detail than last time. As I do not intend to either support inheritance nor use dynamic resources the compiler generated destructor will be fine.

Before I handle the constructors I want to tackle a utility function to reduce a `Rational` to canonical form (i.e. the lowest possible terms).

```
void Rational::simplify()
{
  if (denominator == 1)
    return;  // a whole number

  // discard whole number part
  integer_type hcf =
         numerator % denominator;

  // handle whole number as rational
  if(hcf == 0)
  {
    numerator /= denominator;
    denominator = 1;
    return;
  }
  integer_type temp = denominator;
  while (integer_type quot = temp % hcf)
  {
    temp = hcf;
    hcf = quot;
  }
  // when you get here hcf exactly
  // divided temp and hence is the
  // Highest Common Factor of the
  // original numerator and denominator
  //
  denominator /= hcf;
  numerator /= hcf;
  return;
}
```

If I remember correctly that is a variation on Euclid's algorithm, taking advantage of the efficiency of computers at doing integer division rather than using the more standard recursion through subtraction. I think it gains by reducing the number of decisions.

Now let me deal with the easier of the two constructors. Its declaration is:

```
Rational(
  long int numerator = 0,
  long int denominator = 1);
```

As I am quite happy to allow implicit conversion from an integer type to `Rational` I do not need to qualify this declaration with the keyword `explicit`. However you should note that there is a possible problem in isolation because if we did not handle it separately this constructor would also provide quite wrong conversions from floating types to `Rational`. Read on a bit.

Also if I later decide to implement a multi-byte integer type I will need to add an extra constructor to handle construction of `Rational` from values of that type.

The definition of this constructor is:

```
Rational::Rational
               (long int n, long int d)
  : numerator(labs(n)),
    denominator(labs(d)),
    negative(false),
    converted(false)
{
  if (d==0) throw (IllegalDenominator);
  // check if the signs of n & d
```

```
  // are the same
  if (n/numerator != d/denominator)
     negative = true;
  simplify();
}
```

How should we provide for accidentally trying to create a `Rational` with a zero denominator? Well we must catch the problem before there is any call to `simplify()` because otherwise we would get a divide by zero generated by the operating system. The easiest way to provide a type to `throw` in this kind of instance is to declare a stateless class nested in `Rational`. As I do not want to do anything but signal the problem this will do:

```
class IllegalDenominator { };
```

That must be about as minimalist as you can get. I bet most of you would not expect such a type, no data, no user written functions, would ever be useful. I hope you now know different. While we are handling problems we might as well add a type to handle the cases when either the numerator or the denominator overflows. So that gives us:

```
class NumeratorOverflow { };
class DenominatorOverflow { };
```

Can you spot a problem? Sometimes you do not really care what has gone wrong, you just want to catch whatever the problem is and then handle it. One way of handling this problem is to use an `enum`. Consider:

```
enum Exceptions
{
  IllegalDenominator = 1,
  NumeratorOverflow,
  DenominatorOverflow
};
```

Now the line in the body of the constructor becomes still works but now it is throwing a value that must be caught with:

```
catch( Rational::Exceptions problem )
```

The only problem with this solution is that you cannot pass information that will identify which instance of `Rational` is in difficulty. So perhaps you are thinking that we need something like this:

```
class Rational
{
public:
  enum Exceptions {
    Unknown,
    IllegalDenominator,
    NumeratorOverflow,
    DenominatorOverflow
  };

  struct Exception
  {
    Rational const * instance;
    Exceptions problem;
    Exception(
         Rational const & inst,
         Exceptions ex=Unknown)
      : instance(&inst),
        problem(ex)
    {}
   // other
  }
};
```

This is seductively attractive until you start to think about possible problems and realise that by the time the exception is caught the instance of `Rational` that generated it might have ceased to exist. Hanging pointers in exception objects are about as bad as you can get. What ideas do the experts have?

Another approach is the construction of an exception hierarchy. This allows for a more fine grained catch mechanism. Something like:

```
struct Exception {};

struct DenominatorLimits :
                    public Exception {};

struct ZeroDenominator :
            public DenominatorLimits {};

struct DenominatorOverflow:
            public DenominatorLimits {};

struct NumeratorOverflow :
                    public Exception {};
```

Now we can write such things as:

```
catch (Rational::NumeratorOverflow)
{ /* process */}

catch (Rational::DenominatorOverflow)
{ /* process */}

catch (Rational::DenominatorLimits)

{ /* process */}

catch (Rational::Exception)
{ /* process */}
```

Remember that in such cases that the most derived exception object must be caught first. I hope you begin to see the range of choices that need to be considered for even something as simple as a `Rational` type. The benefit is that when you have invested time doing a robust design you have a solid class that can be used again and again. Good design can be developed. Inexperienced programmers find well designed classes easy to use. If you find that you need to know a lot about the inner workings of a class or that its use springs surprises on you then the class is probably badly designed. Despite all that is said about Java, it has just the same design problems and probably has fewer tools for solving them. That, of course, is a matter of opinion but at least think about it before discarding C++ as being too difficult and adopting Java because you think it is easier.

Now let me consider one more item (plenty left over for next time). We need to urgently deal with the second constructor, the one that takes a floating point type as a parameter. The problem that we have is that we have to convert a `long double` into a `Rational` and not all will work. Perhaps we need to introduce something like the IEEE concept of NaNs (not a number). We could do this easily by adding an extra data member. Do we want to just track `Rational` instances that are simply invalid or do we want to do more? Let's keep it simple for now and just track validity. Add `bool valid;` into the data for a `Rational`. Back track to the constructor we already have and add an appropriate initialiser into the constructor-initialiser list. Also set `valid` to `false` before throwing an exception.

```
Rational::Rational(long double lf)
  : converted (true),
  fp_value(lf)
{
  if (lf < 0)
  {
    negative = true;
    lf = -lf;
  }
  else
  {
    negative = false;
  }
  //
  //note this is implementation dependant
```

```
  //
  if (lf > UINT_MAX)
  {
    valid = false;  // mark as not representable
    numerator = 1;  // and place in stable state
    denominator = 0;
    throw (NumeratorOverflow)    // or whatever you are using
  }
  integer_type whole = lf; // save the non-fractional part
  long double fraction = lf - whole;
  numerator = fraction * 65520;  // see note below
  denominator = 65520;
  // simplify the fractional part as a Rational
  simplify();
  long double check = denominator * static_cast<long double>(whole) + numerator;
  //
  // reduce numerator to available range
  //
  while(check>UINT_MAX)
  {
       check /= 2;
       denominator >> 1;
  }
  numerator = check;// and convert it back to an integer
  return;
}
```

I would be the first to admit that this is a pretty lousy conversion. This is exactly why programming teams need implementation specialists who have a wide knowledge and understanding of algorithms. What we need is the pair of integers that most nearly represents the `long double` value passed in. The above algorithm will badly miss. For example it will get nowhere near the good rational representations of π that are known. The use of 65520 as a scaling factor is because this is close to the maximum 16-bit `unsigned int` limit and has a substantial collection of factors that increases the chance that `simplify()` will reduce the magnitude of the denominator and hence reduce the need to further approximate. It still includes repeated factors of two which reduces the likelihood that the scaling required by large non-fractional parts will even further reduce accuracy. None the less I am far from happy and would be delighted if some numerical expert can come up with a better algorithm. What is particularly unpleasant is the degree to which this function depends on the true type of `integer_type`.

Well I think this is enough for this time. Please feel free to criticise and to add your suggestions.

The Harpist

Some Opportunities to Increase STL Efficiency
By Sergey Ignatchenko

As the Standard Template Library (STL), part of the Draft C++ Standard, becomes more and more popular among programmers, some interest will be focused on its performance characteristics. This article is devoted to the analysis of ways to increase the efficiency of the STL. Both the offered solutions are fully compatible with existing STL versions, and may provide a ten-fold performance gain.

The Existing Problem

In some collection operations it is necessary to move objects from one memory location to another. The STL creates a new object via the copy constructor, and deletes the original. If the object being moved is non-trivial, considerable unproductive time may be wasted in the construction and destruction of the objects.

Take for example the vector collection class. The vector is implemented as a contiguous

memory block. This approach has its advantages, but its serious disadvantage is that inserting or deleting elements in the middle of this memory block may take considerable time.

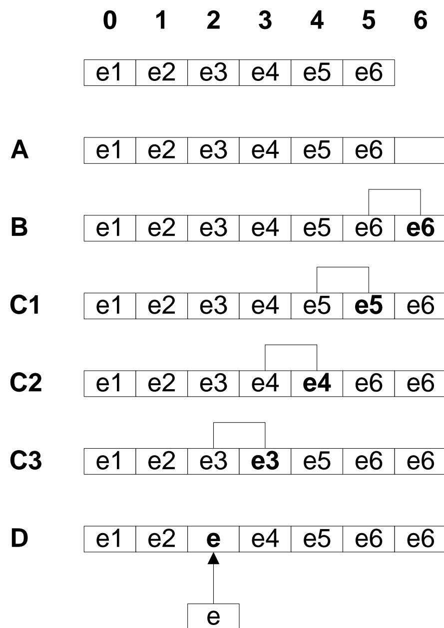
Consider a sample vector consisting of six elements *e1-e6* of type E, accordingly placed in positions 0 to 5. In order to insert an element in the middle of the vector, say position 2, the STL implementation must:

A: provide space for 7th element;

B: create in position 6 an element, which must be a copy of *e6* (using copy constructor);

C1-C3: move elements *e5-e3* one position to the right (using assignment operator);

D: assign value *e* to the element in position 2.



It is obvious that two copies of e6 element exist between steps **B** and **C1**, and one of these copies (in position 5) will be deleted immediately. The same happens during steps **C1-C3**: copies of *e5-e3* elements are made, and the original elements are immediately deleted. It has little effect if type E is trivial (*int*, for example), but if type E is complicated enough (*vector<vector<int>>*, for example), two serious problems may arise:

1. Both copies of one element use memory, and if the objects are large an out-of-memory problem may occur.

2. Copy element creation and original element deletion take time and may cause an essential increase in program execution time.

The above problems are typical not only for vectors, but also for some algorithms, such as *remove* and *remove_if*.

As a simple class example, which has nontrivial copy constructor and assignment operator, let us consider the *PseudoString* class:

```
class PseudoString
{
  char* s;

public:
  PseudoString()
  {
    s = 0;
  }
  PseudoString( const char* str )
  {
    _init( str );
  }
  PseudoString(
           const PseudoString& other )
  {
    _init( other.s );
  }
  void operator =(
           const PseudoString& other )
  {
    if( this != &other )
    {
      delete [] s;
      _init( other.s );
    }
  }
  ~PseudoString()
  {
    delete [] s;
  }

private:
  void _init( const char* str )
  {
    if( str )
    {
      s = new char[ strlen( str ) + 1 ];
      strcpy( s, str );
    }
    else s = 0;
  }
};
```

Move Instead of Copy

One of the possible solutions to overcome the above problem is to change the STL implementation by replacing the use of the copy operation for that of a move operation, which in most cases can be implemented more efficiently.

Let us consider the *assign_move(E& a, E& b)* function, which is essentially a "move operator". If after applying it to objects A and B the following conditions are valid:

- object A becomes equal to the original value of object B;
- object *B* remains a correct object of type E, whilst the actual value of object *B* is not meaningful.

The "move constructor", implemented as the construct_move (E* a, E& b) function is defined in similar way.

To benefit from the replacement of copy operations with move operations the following changes to the STL are required:

1. Define template functions *construct_move* and *assign_move, which calls* copy constructor and assignment operator accordingly:

```
template <class T1, class T2>
inline void construct_move(T1* p,
                           T2& value)
{
  construct( p, value );
}

template <class T1, class T2>
inline void assign_move(T1& a, T2& b)
{
  a = b;
}
```

2. Introduce functions *move/move_backward/ uninitialized_move* similar to *copy/ copy_backward/uninitialized_move* but based on *construct_move* and *assign_move*:

```
template <class ForwardIterator,
          class OutputIterator>
OutputIterator move(
              ForwardIterator first,
              ForwardIterator last,
              OutputIterator result)
{
  while (first != last)
    assign_move(*result++, *first++);
  return result;
}
```

3. Replace *copy/copy_backward/ uninitialized_copy* calls, with *move/ move_backward/uninitialized_move* calls, in the situations similar to mentioned above:

```
void erase(iterator position)
{
  if (position + 1 != end())
    move(position + 1, end(), position);
  --finish;
  destroy(finish);
}
```

A programmer using the above modified STL is now able to define, apart from traditional copy constructor and assignment operator, their more efficient "twins": *construct_move* the "move constructor" and *assign_move* the "move operator", for example:

```
class PseudoString
{
  ...
public:
  void swap( PseudoString& other )
  {
    ::swap( s, other.s );
  }
};

inline void construct_move(
  PseudoString* p, PseudoString& value )
{
  new( p ) PseudoString();
  p->swap( value );
}

inline void assign_move(
      PseudoString& a, PseudoString& b )
{
  a.swap( b );
}
```

If a programmer uses this opportunity he may get a considerable reduction in program execution time.

Otherwise the calls of *construct_move*/ *assign_move* functions correspond to calls of their template versions, which, in turn, are equivalent to calls of copy constructor/ assignment operator. Thus, a program using modified STL will operate just like the program using original STL.

Bitwise Move

Let's continue with our performance improvements. By doing without class E copy constructor and assignment operator and using a bitwise move of memory block instead. This is possible for most, but not for all classes.

Let's consider class X

```
class X
{
  X* x;

public:
  X()
  { x = this; }

  X( const X& other )
  { x = this; }

  void operator =( const X& other )
  { x = this; }

  void f()
  { assert( x == this ); }
};
```

This class contains data member *x*, which is always (including the case when it is being copied/assigned) equal to *this*. It is obvious that in case of bitwise move this equality will be broken and an error will occur. Thus the possibility of bitwise move depends upon class E nature.

It is necessary to make the following changes in the STL in order to use bitwise move for vector optimization:

1. Define a *can_bitwise_move* function template, returning false.

2. Use bitwise copy where possible, provided that for the class, contained in the vector, *can_bitwise_move* returns *true*.

After that a programmer who uses modified STL is able to define *can_bitwise_move* function which returns *true*, for class E, and let STL use bitwise move for this class. Here is an example of implementation of this function for class *PseudoString*:

```
inline bool can_bitwise_move(
                   const PseudoString* )
{
  return true;
}
```

"Bitwise move" optimization is more efficient for vectors, than "move instead of copy" optimization. On the other hand "bitwise move" optimization (contrary to "move instead of copy" optimization) can not be used for algorithms.

Optimization

Three implementations of STL libraries were used for optimization:

1. Hewlett-Packard STL (1994), (ftp:// butler. hpl. hp. com)

2. Silicon Graphics STL (1996), (http:// www.sgi.com/ Technology/ STL/)

3. adaptation of Silicon Graphics STL by Moscow Center for SPARC Technology (1997), (http:// www.ipmce.su/ ~fbp/ stl/).

Optimized versions of these STL implementations are available at **ftp://....** . Besides, Microsoft STL implementation supplied with Microsoft Visual C++ 4.2 was used for check timing.

Both above methods of STL optimization ("move instead of copy" and "bitwise move") were implemented. A number of class *vector* member functions (insert, erase, reserve) were optimized. Besides, some algorithms (*remove*, *remove_if*, *unique*) were optimized using "move instead of copy" method.

Optimization "move instead of copy" is on by default; macro definition *__STL_NO_MOVE_INSTEADOF_COPY* should be used to disable the optimization. Optimization "bitwise move" is off by default; macro

__STL_BITWISE_MOVE should be defined to enable the optimization. If macro *__STL_NO_MOVE_INSTEADOF_COPY* is defined and macro *__STL_BITWISE_MOVE* is not defined, then optimized STL is identical with the original STL.

Comparative Efficiency Analysis

Two test programs - *testmax* and *testrnd* were developed to compare efficiency. Both programs were compiled with Microsoft Visual C++ 4.2 compiler in "Release" configuration. Check timing was made on Pentium 100 with 32 MB RAM running under Windows NT 4.0 operating system.

testmax program was developed to demonstrate the maximum gain of optimization. The program constructs vector, adds 1 complicated element (E(M)), and then inserts simple element (E(0)) in the beginning of the vector K times. Here M describes the complexity of element.

testmax **Execution Time, ms**

M	MS	HP	SGI	MOVE	BIT
16	400	1400	1400	70	40
32	600	1600	1550	75	40
64	1400	2400	2500	80	40
128	3350	4000	4250	85	45
256	21700	21200	25200	95	60
512	160000	132000	140000	180	150
1024	410000	430000	440000	440	400

MS Microsoft STL

HP Hewlett-Packard STL

SGI Silicon Graphics STL (both original and adapted)

MOVE STLs optimized by "MOVE INSTEAD OF COPY" method

BIT STLs optimized by "BITWISE MOVE" method

The gain in program speed, resulting from optimization, depends on element complexity and varies from 10 to 1000 times.

The second program - *testrnd* - was developed to estimate the average gain resulting from optimization in some standard case. The program constructs a vector consisting of *PseudoString*-type elements and makes random manipulations with this vector (mostly insertions and deletions) NRND times.

testrnd **Execution Time, ms**

NRND	MS	HP	SGI	MOVE	BIT
512	260	185	185	40	30
1024	630	460	470	70	50
2048	1350	1000	1050	150	100
4096	2700	2000	2000	290	210
8192	20000	15500	15500	1000	500
16384	26000	19700	20000	1450	850
32768	126500	97500	97500	5300	2100

The gain in program speed depends on NRND and varies from 5 to 50 times. Thus the testing showed that the above STL optimization methods may enable considerable gain in program speed, and in special cases this gain can exceed 1000 times.

Compatibility

Although both described methods of STL optimization imply interaction with the programmer using it, they are 100%

compatible with the original STL. That means that a correct program developed for original STL will always work with optimized STL, and vice versa. It is also obvious that the program speed is improved only when two following conditions are met:

1. The optimized STL is used;

2. The program supports *construct_move*, *assign_move* and/or *can_bitwise_move* functions for some classes (the program is STL optimization-aware).

	STL optimization non-aware	STL optimization aware
Original	+	+
Optimized	+	**FAST**

1. + program operates at regular speed

2. **FAST** program operates faster due to STL optimization

Summary

The key to the technique described here is in interaction between STL writer and programmer using it. Use of this technique is not limited to methods of optimization described in this article. On the other hand, these methods can be used to make some other optimizations, such as optimization of *deque*s. Thus there are still lots of opportunities to increase STL efficiency.

Sergey Ignatchenko
ignatch@rtsnet.ru

editor << letters;

In Defence of Standard C++

From The Harpist

I found the item from Alan Griffiths in Overload 21 disturbing. When I coupled that with your open letter about the future directions for Overload I felt that I had to respond.

I have been around C++ for almost a decade and during that time I have watched it develop from the language of the first edition Bjarne Stroustrup's 'The C++ Programming Language' through that of the second edition into that of the third.

In the first edition we have, basically, an extended C. The extensions produced a much nicer tool for high-level development, but fundamentally the modifications and book were primarily addressed to C programmers. The language was young and had yet to develop its own native idioms. At that stage a good working knowledge of C was a pre-requisite for mastery of C++.

By the second edition certain aspects of the language were beginning to gel. However the language was at the stage where all its inner workings were exposed even to the novice. While the language enabled implementation hiding etc. the user was expected to know far too much. At this stage we began to hear rumblings about how complicated C++ was. The disgraceful inadequacies of many writing books for newcomers and providing training only made matters worse. We had situations were programmers were learning C++ one week and presenting training courses on it less than three months later. We had authors of poor books on C rehashing their work and selling the results as books on C++. This is still a major problem today.

With the third edition we begin to see a mature product where much of the complexity is being encapsulated. We still have serious problems with our development tools. The diagnostic messages being produced by leading products look like garbage. However it is possible (though trainers in general have yet to catch up) to introduce a non-programmer to C++ without any of the mystifying mumbo-jumbo that has made programming so difficult in the past. Of course if you insist upon using char * for strings and manually iterating across an array used as a container your code will be difficult, bug-ridden and hard to maintain. The point I want to make is that it does not have to be that way. Compare the controls of a modern jet airliner with those of a prop driven aircraft of the 50's; the complexity is now hidden, and information is provided on a need to know basis. The same contrast exists between the C++ of 1985 and the best C++ of 1997. Unfortunately we not only have millions of lines of legacy code, mountains of ill-conceived poorly written C++ books, thousands of inadequately trained C++ programmers but we also have managers looking for that magic cure-all.

As a result we are now producing millions of lines of badly written Java, mountains of ill-conceived poorly written Java books and tens of thousands of inadequately trained Java programmers. We also, for a brief moment, have managers who, tired of the long time it has taken to develop C++ and continual complaints about how difficult it seems to have become, believe this brand new shiny object will magically cure their problems.

One task that the C++SIG should take on board is that of ensuring that we have a clear grasp of what C++ is and what it can achieve so that we do not scurry off to the newcomer

on the block in the home that it will solve all our problems for us.

OK, so I do not agree with Alan. Now what about the future directions for Overload? I believe that lead by such contributors as Francis with his love of the arcane, Overload managed in the past to miss the needs of the serious C++ programmer. The first thing we should be clear about is that Overload is the voice of the C++SIG. If you want a Java SIG create one. Just as C Vu covers elementary aspects of C++ (and Java) Overload can, and should cover elementary aspects of languages related to C++. Members should not bury their heads in the sand and ignore Java, Objective C, Smalltalk, Eiffel etc. But the focus should be 'what C++ programmers need to know about these things.' Articles on how to interface Java with C++ would be fine. Articles on the criteria for choosing Java, Eiffel or whatever instead of C++ are also fine but an article on Java programming out of the context of C++ would, in my opinion, be stretching it.

I want to see articles on other aspects apart from pure coding. Analysis and design methodologies are an important aspect of being a C++ user. I would love to read articles about testing, multithreading, parallel processing etc. Articles about patterns are fine but I would prefer to see them focus more on the intermediate and low level (idioms) than on the large scale Design Patterns that are characteristic of TgoF's book. As many members are less than familiar with patterns at any level I would be delighted to see a regular spot where a design pattern was explained in simple terms.

We should bare in mind that there are at least three separate types of C++ programming: class design, class implementation and application programming. Each of these requires a different set of skills and insights. Only the rarest programmer will be good at all three, however Overload should be addressing the needs of all of them. While I can do a fair job at designing a single class, I am much less able at class hierarchy design (I would love to read a series on that). I am reasonable at class implementation and only the kind would describe me as even capable when it comes to application design and implementation. Do not even mention analysis to me, I have no idea about how to do it.

I think that the readership would be far happier to see Overload focus on helping them to write better C++ than following the latest over-hyped language trend. Of course we have to remember that the writers are a sub-set of the readership so when it comes down to it we will only get what they choose to write about.

The Harpist

More Defence of Standard C++

From George Wendle

I was amazed at Alan Griffiths' article in Overload 21. I do not think that there was much choice over when the C++ standardisation effort should start even though the language was only half complete in 1990. The support for exceptions, generic programming etc. was not some fancy new idea but an integral part of the language as conceived by its parent, Bjarne Stroustrup. A few things were introduced in addition to the original design and some requirements were relaxed (the return type of polymorphic functions). Support for type information was standardised only after numerous libraries had generated their own work rounds. Namespaces were introduced to fix growing problems with name collisions and the new casts were introduced because the old C sledge-hammer simply could not cope. The Standard Template Library was not an addition to the language but it was the single most dramatic enhancement made in the last decade. It may not be perfect but it has made C++ dramatically more useable by

ordinary programmers (if only people would explain it to them).

At last we have C++ stabilising and maturing into a language that meets its original promise of addressing the needs of a wide community. So what does Alan advocate? We all jump for Java while it is still in a state of extreme flux. I can think of nothing worse than to take people who have just begun to understand modern C++ and then dump Java on them. That is a recipe for total confusion. They will write something that is simultaneously bad C, bad C++ and bad Java. Programmers who already struggle with the semantic differences between references and const references are not likely to grasp the subtle difference between built-in and class types in Java. Java variable names look like C ones but are entirely different. I can think of no bigger potential for disaster than to take a befuddled C++ programmer and move them to writing Java to mix with existing C++.

Those of us who live through change see far more difficulties then those that start at the end. The C++ as described in 'The C++ Programming Language, 3rd ed.' seems to be an excellent tool. I hope that before jumping for the next fad, most of you will learn the idioms of modern C++ and stop complaining about complexity brought about by writing code at an unnecessarily low level.

George Wendle

Overload Content

From Francis Glasborow

Judging by the two letters that have been forwarded through my mail box, Alan Griffiths' item in the last issue of Overload has touched on a couple of sensitive nerves. I think that the issues need airing but I also think that we should think carefully about what this SIG is. In the past I have published the occasional article in C Vu on such things as awk, assembler programming etc. but I am pretty sure that the membership would get pretty annoyed if I started a regular section on such things as Visual Basic or Delphi. They would probably be happy if I published the occasional informative article about what these have to offer, or something about how to access a DLL written in C/C++ from them but there is definitely a boundary all be it an ill-defined one. It is part of the job of an editor to draw the line. You can be a conservative editor who always stays well within the remit or you can stick your neck out and get thoroughly reprimanded when you over-step some reader's concept of where the boundary lies.

Java (and Objective C, the various parallel C's, Python etc.) is more problematical because they are close relations. If we could get contributors on Objective C, I am sure that such material belongs in either C Vu or Overload. Material on parallel C's definitely belongs in C Vu and I would be delighted to be able to publish regular material on these (one way or another they will become increasingly important in the future).

Python is interesting and deserves greater exposure because that language has a number of interesting features which include excellent facilities for mixed programming with C and C++. Again the problem is the lack of someone who is willing to write a regular column.

That leads me to Java. It looks like C/C++. It certainly should interest users of C++. However it has been seriously over-exposed recently and I suspect that its problems are only just beginning. One problem that should not be shoved under the carpet is that it was not designed for traditional development where the programmer uses a text editor. The whole monolithic structure assumes that you will have appropriate tools to present alternative views of the same material. I am also unhappy with writing debugging/testing code for Java. That may just be me but I suspect that there are serious problems of scalability awaiting exposure.

I have yet to see anyone write native Java (to borrow an expression from the Harpist's recent C Vu item). I think that most code is still thinly veiled C or C++. C++ used to write C usually results in both poor C and poor C++--look at most of the dozens of introductory books on C++. There is still work to be done on the Java language itself. Possibly these developments will be small, but I am not convinced that this will be the case. Many thought that the developments of C++ would be small (had they been so and the work had stopped where C++ was in 1989 we would have had a language of passing interest) but in the event much work was needed to consolidate and complete the language. Alan comments on the standardisation of C, actually from the insider's perspective much more had to be done than appears on the surface. Some of the 'simplicity' was bought at a price of ruthlessly postponing some developments. I think that the next C standard will pay a very high (I am still trying to decide if that should be unacceptably high) price to retain the apparent simplicity. Actually it is immensely difficult to write fully conforming C and most C programmers completely ignore the multitude of test macros and the like that are required to write C that can be reliably ported from platform to platform.

I think that Java should very definitely be classified as an experimental language for some time to come. I would be happy to see articles on it in both C Vu and Overload but I would still want to see our major focus elsewhere.

What I would very much like to see would be articles that explored how a selection of algorithms and patterns would be naturally coded in a range of languages from the C family. Let me set a couple of tasks. How would you implement?

1. A lottery number generator (x numbers from y choices, with possible constraints if you want to be ambitious)
2. A search of a text file for a match of a text string (including wild cards if you are ambitious).

in the language(s) of your choice. It would be interesting if you also provided your criteria for language choice. Also do not feel constrained to limit yourself to languages I have mentioned in this letter.

What would be nice is if every reader spent a little time on one of these problems and submitted something. Of course that won't happen but how will you feel when you see dozens of contributions next time and, too late, you realise that you could have done it better, differently or...?

Francis Glassborow
francis@robinton.co.uk

Finally Support for Alan's View

From Chris Southern

I must start with an expression of fellow feeling for Alan Griffiths.

Every time I try to get a handle on the first stage of re-use in C++, i.e. the Libraries, the rug is smartly whipped from under my feet.

Usually I manage to buy a decent book on part of the current idea of the libraries only for it to be outdated in very short order. Sometimes it was outdated when I bought it, but the articles telling me the new one true way have been in the LONG pipeline of publication to C++ Journal etc.

How stable is it and when are we going to see helpful reference works on it?

Now an example by way of illustration of 'helpful' I recently needed to prepare some formatted data to be returned by a member function. Being a dutiful observer of the axiom about wheel re-invention and having been abjured to wean myself off `char*` I wanted to put output into a String. The first part of the Metrowerks Code Warrior documentation on the 'Standard' Library concerning this, showed two constructors for a `basic_ostringstream< charT, char_traits<charT>>` one with only a mode parameter, and one with an additional String.

With syntax, but no semantics given, I foolishly thought that the second constructor must be just the thing, obviously it attaches a stream to a string, and I could pass the string back later. In my defence I cite the reference itself: *"the basic_stringstream constructor is overloaded to accept a an object of class basic_string for output."*

This is completely in defiance with the signature of the constructor, which is:

```
explicit   basic_ostringstream
(const     basic_string<charT>
&str, ios_base::openmode which
= ios_base::out)
```

note the `const`! The thing actually uses the input string to initialise the completely separate internal string used as a buffer.

This is obviously a definition of unhelpful.

Chris Southern
csouthern@brasspaw.cix.co.uk

Readership Feedback

From The Harpist

Well, nothing has been forwarded to me from my previous 'Rational Type' article, which I find a bit disappointing. Lack of response generates a mixture of reactions within me. Perhaps the material was to simple and left readers feeling bored. Perhaps it was badly written so that most did not understand what it was about. Perhaps the idea of a complete design was simply beyond readers who had time to do anything. I simply do not know. What I am certain of is that some interaction is necessary. I was talking to Francis recently about his regular column in EXE Magazine and he told me that he rarely gets more than a couple of letters, emails etc. about any column even when he has made a serious mistake. He finds that despite being paid for his column he would still like a response. All the more is that the case when the effort is entirely voluntary. I wonder how you would feel if you seemed to be completely ignored? Imagine yourself presenting an item at a conference and when you had finished everyone just got up and walked out. Would you feel like doing anything else for such an event?

By the way, that reminds me that I should be thanking Francis and Parkway Gordon for the superb event they put on. The organisers should feel very pleased with themselves for having created such an enjoyable and informative event. I attend quite a lot of conferences and it is rare to find one with quite such a buzz. Those of you who went certainly understood the value of talking to each other. I particularly appreciated the way that so many of the speakers joined in with the spirit of the event. I hope that many of you will flood Francis with ideas for next time. Changing the World is hard work and needs teamwork.

Pages of letters responding to material published in Overload would be nice and would certainly encourage the regular contributors to keep going.

The Harpist

ACCU and the 'net

ACCU.general

This is an open mailing list for the discussion of C and C++ related issues. It features an unusually high standard of discussion and several of our regular columnists contribute. The highlights are serialised in *CVu*. To subscribe, send any message to: accu.general-sub@monosys.com

Demon FTP site

The contents of *CVu* disks, and hence the code from *Overload* articles, eventually ends up on Demon's main FTP site: ftp://ftp.demon.co.uk/accu Files are organised by *CVu* issue.

ACCU web page

Thanks to Net Access and DeMontfort University we now have a machine permanently connected to the Internet. The official ACCU web pages have moved to a new home. http://www.accu.org/

C++ – The UK information site

This site is maintained by Steve Rumsby, long-serving member of the UK delegation to WG21 and nearly always head of delegation. http://www.maths.warwick.ac.uk/c++

C++ – Beyond the ARM

Sean Corfield maintains a set of pages about recently added C++ features. He welcomes feedback on their content. http://www.ocsltd.com/c++

Contacting the ACCU committee

Individual committee members can be contacted at the addresses given above. In addition, the following generic email addresses exist:

caugers@accu.org,	chair@accu.org	cvu@accu.org
info@accu.org	info.deutschland@accu.org	membership@accu.org
overload@accu.org	publicity@accu.org	secretary@accu.org
standards@accu.org	treasurer@accu.org	webmaster@accu.org

There are actually a few others but I think you'll find the list above fairly exhaustive!

Beyond ACCU...*UseNet*

This small section will highlight other internet resources that are relevant to Overload readers.

UseNet groups have variable quality. The moderated news groups (eg.. comp.lang.c.moderated, comp.lang.c++.moderated) are easy to monitor. The other groups are harder work *but* they tend to have summaries of "frequently asked/answered questions" (FAQs), posted by volunteers to each newsgroup. Web browsers can be used to get UseNet FAQs.

http://www.lib.ox.ac.uk/internet/news/faq/by_group.index.html.

It you have a useful link, share it! (Please send it to new-links@accu.org).

Credits

Editor

John Merrells
merrells@netscape.com

4 Park Mount,
Harpenden, Herts, AL5 3AR,
U.K.

1111 El Camino Real #109-264,
Sunnyvale, CA 94087,
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

Subscriptions

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

Copyrights and Trademarks

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

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

Copy deadline

All articles intended for inclusion in *Overload 23* should be submitted to the editor, John Merrells < merrells@netscape.com>, by November 14th.