contents

credits & contacts

Overload Editor:
Alan Griffiths
overload@accu.org
alan@octopull.demon.co.uk

Contributing Editor:
Mark Radford
mark@twonine.co.uk

Readers:

Ian Bruntlett
IanBruntlett@antiqs.uklinux.net

Phil Bass
phil@stoneymanor.demon.co.uk

Thaddaeus Frogley
t.frogley@ntlworld.com

Richard Blundell
richard.blundell@metapraxis.com

Advertising:
Chris Lowe
ads@accu.org

Overload is a publication of the ACCU. For details of the ACCU and other ACCU publications and activities, see the ACCU website.

ACCU Website:
http://www.accu.org/

Information and Membership:
Join on the website or contact

David Hodge
membership@accu.org

Publications Officer:
John Merrells
publications@accu.org

ACCU Chair:
Ewan Milne
chair@accu.org

Editorial: New Things Under the Sun

Mark's editorial "An Industry That Refuses To Learn" (Overload 60) clearly struck a chord of recognition – as you can see from the letters page. It is disappointing, however, that no-one makes the case for there being improvements in the industry. Despite the depressing repetition of mistakes and rediscovery of ideas, some things have changed for the better!

One of the things that has changed is the ease with which ideas, techniques and even tools circulate. And this isn't just the advent of new technologies – when I started in the industry the only books around the office tended to be manuals for the system being used. Now I can look around the office to find copies of DeMarco's "Peopleware", Senge's "Fifth Discipline", Cockburn's "Effective Use Cases" and "Agile Software Development", along with a miscellany of books on Design Patterns and Extreme Programming. Some of the ideas these contain are even being put into practice! It may be taking time, but change is happening. And Overload has a role to play in ensuring that useful knowledge is not forgotten.

Of course, the trouble with change is ensuring that we keep the good and discard the bad. And that is particularly on my mind at the moment. The more observant of you may have noticed a change to Overload (but I hope that there is nothing that has caused you to notice). Let me explain.

At the AGM Tom Hughes stood down from the post of Publications Officer having successfully steered the journals through a number of changes in the last few years. This gave the previous Overload editor, John Merrells, the opportunity to join the committee as the new Publications Officer (which might be considered a promotion). That left a gap to be filled as Overload editor – and, when it was offered to me, I took it on. I hope that you will all join me in thanking Tom and John for their efforts over the years, and to wish John success in his new role.

It was only a year ago that I joined the Overload team as a "contributing editor" – and took on the some of the editorial tasks, such as producing the editorial. These are tasks that I've subsequently shared with Overload's remaining contributing editor, Mark Radford. The first of these editorials reviewed the changes that had happened to Overload since John took it over (from me) and concluded: "All of this makes Overload a much more impressive publication than it was six years ago."

The last year has not changed that opinion: I'm very happy with the way that Overload has developed under John Merrells' guidance. I hope that you are too. It should therefore be no surprise that I've not taken on the editorial role with any dynamic agenda for change. Most of the Overload team remains in place and will be performing the same role: helping authors to prepare their work for publication. And with any luck you, the readers, will continue to find interesting things happening to write articles about.

Having said that, I must immediately confess that some changes are intended: John Merrells, Paul Johnson (the new C Vu editor) and I have agreed that the distinction between the journals has not been clear to the readership. In part this has been due to a failure to route articles to the other journal when that was appropriate. In the future we intend to ensure that articles are re-routed if appropriate. I trust that this won't upset anyone.

Which all begs the question "in which journal does an article belong?" While those working on the journals tend to agree on each specific article that has been discussed, it isn't that easy to explain. Especially as some members (and the ACCU website) appear to have views that differ markedly from ours. So let me go on record about a couple of things:

- Overload is no longer a C++ based publication: it covers a much broader range of material – the current issue discusses team working; design patterns; the use of exceptions in C#, Java and C++; and the high competency threshold set by C++.
- Overload is not a minority interest: the vast majority of ACCU members (just over 85%) subscribe to Overload.

So what is Overload about? In the editorial mentioned above I said of Overload articles that "there is a tendency for them to be about designs, illustrated using C++, rather than about C++ itself". Having thought about it over the past year I no longer think that captures it at all: The articles assume that most basic development skills have been mastered and address problems faced by journeyman (or master) developers. (While the industry does not have a recognised apprenticeship system it does have recognised masters.) These articles might examine the effect of applying alternative solutions; or discuss new tools and ideas; or show old ideas in a new light. There are reports back from those venturing away from the mainstream approaches in the hopes of finding something better. And don't forget that what is "mainstream" for one technology may be new ground for another – we need to exchange ideas.

And the exchange of ideas was the theme of a "Birds of a Feather" [BOF] session on C++ that I led at the conference. (Why C++? – because I'm currently working in that language and missing the support that is available for working in Java.) In that BOF I led a discussion of tools and techniques that people had enjoyed in other languages that they lacked when working in C++. Many things came to light from developers using Java:

- Using a refactoring IDE (Eclipse, IntelliJ) changes the way that you think about maintaining and developing code (and makes things achievable that were not before).
- Having a de-facto standard for a unit test framework (JUnit) makes test driven development that much easier to establish.
- Tools for automating integration, build and smoke test processes (CruiseControl, AntHill) make frequent (or continuous) builds easy to put in place – not a continuous battle.

I'm sure there are others, but these types of tool have a long history in the SmallTalk community – and only became commonplace when ex-SmallTalkers reimplemented them after moving to Java.

The BOF wasn't entirely a matter of discussing the greener grass and sighing with regret: an interesting discussion of various unit testing frameworks developed with people relating why they'd written their own (about half the room had done this). People expressed various dissatisfactions with `CppUnit`, `CricketTest`, `CppUnitLight`, and `CxxUnit`, but, interestingly, `boost::test` got no bad reports. There were not enough reports on `boost::test` to announce it a winner, but people were sufficiently interested to say they'd go away and try it. (I've not had any reports back yet – but I'll keep you informed.)

Another thing that was discussed was whether there were technical obstacles to the development of C++ refactoring tools. It was felt that the difficulty of compiling C++ and the way that a piece of code can appear in multiple translation units made this a lot harder than equivalent tools for SmallTalk or Java. But it was thought possible.

Developers, it seems, are not the only ones to recognise the potential for refactoring tools for C++. After I mentioned refactoring tools for C++ in my "Christmas List" (Overload 58), Giovanni Asproni emailed me with links to an alpha version of a C++ refactoring tool: `http://www.xref-tech.com/` and to SlickEdit: `http://www.slickedit.com/` which advertises these features. Microsoft has also listed "refactoring" as a feature of their forthcoming IDE. I've not, as yet, had time to review any of these offerings, so the reality may be less than the promise, but I am looking forward with anticipation.

Of course, it isn't just within the C++ arena that things are happening. I've been hearing from developers taking a first look at Java's JDK 1.5, which it seems has grown a host of new features. But while I've heard a few rumours I can't yet provide an informed report of any interesting developments there.

I'd love to be able to report on progress in other areas:

- Languages like Python and C# are widely used by ACCU members. (But not, at the moment, by me).
- New development tools have become available and some of them change the way we can think about developing software. (I've mentioned some of these above.)
- Ideas and techniques for handling various elements of the development process are being explored. (I know of one developer who took the suggestion that "test driven development wouldn't be appropriate to a compiler" as a challenge, and has used that approach to write a compiler.)
- Anything else that develops the art/craft/science of software development.

I want to know more, and my guess is that many of you do too. I also know that my time is limited and have no reason to believe that yours is any different. But if you are reading this and thinking "doesn't he know about X" then you can help the rest of us by submitting an article. (It is only fair, you discovered about A, F and M through articles in Overload.)

C++ wasn't the only thing I talked about at the Spring Conference – I ran a second BOF in conjunction with Neil Martin that discussed the possible role that ACCU could play in promoting a professional approach to software development. Lots of ideas were discussed, including certification schemes. Neil went away to investigate what the implications of becoming a certifying body were, and I got a mailing list set up that includes the people that expressed an interest. While Neil has reported back to the list nothing else has happened (yet). Perhaps someone reading this can provide the necessary impetus.

I hope that I'm going to enjoy editing Overload, I have certainly discovered that it involves work. I'm writing the first draft of this editorial on the train home from working at a client's for the last few days – and it has to go to the production editor tonight! (So no chance to mull it over for a few days.) But, so long as worthwhile articles are submitted, I feel that the effort is worthwhile.

Things are changing: new people, new ideas, some things don't – some problems are there just because there are people involved. But let me repeat a suggestion made to me at the Extreme Tuesday Club (sorry, I've forgotten your name, but I owe you a pint) of something that isn't just an old idea reinvented: aspect oriented programming. What do you think? Seen it before?

While Mark had a valid point that some good old things have been forgotten, there are sometimes new things under the sun, and these may be good too.

Alan Griffiths
`alan@octopull.demon.co.uk`

Copy Deadlines

All articles intended for publication in *Overload 62* should be submitted to the editor by July 1st 2004, and for *Overload 63* by September 1st 2004.

Letters to the Editor(s)

Software Project Management Classics?

Dear Mark

Your Overload 60 editorial was very timely for me. An open-source software project on which I am involved is just getting going, and it is clear that good communication between participants is of paramount importance. I'm also rather unimpressed by the project management tomes that line the shelves of today's bookshops. I have a question, arising from your editorial, that I'd be interested in pursuing via the ACCU, but I'm not sure which would be the best forum.

First some brief background: I'm an ex-academic, just over halfway through a career change into software development. I have only a little formal education in computer science and virtually none in project management.

Frederick Brooks' book "The Mythical Man-Month" was one of the first project management books I read, and it still stands head and shoulders above the doorstops. Not only do I re-read Brooks, but many of his ideas have taken root in my head (I like to think); the doorstops are in general eminently forgettable.

I completely agree with you on the waste exemplified in your story about your friends from Bucks: knowledge sent out to fallow, and new generations treading the same ground blindly. This kind of amnesia is surely unimaginable in any other field (not even pop music is so forgetful).

Is there a good list of 'classics' of software project management, like Brooks' essays? If not, I'd be interested in doing a straw poll of ACCU members and keeping a tally linked from my homepage (`http://www.iau.ukfsn.org`).

Yours sincerely

Ivan Uemlianin

The Invisibility of Software Design

Mark

I read your editorial in April's Overload with a sense of depressing agreement. I have been in the industry for around ten years and I have worked in most phases of the software development cycle, from requirements through to some post-installation customer support. I have worked as a programmer, a designer and as a consultant advising companies on development strategy and technology policy. I think that I am just getting to the point were I can understand what is going on and might have something to say about it.

I have been appalled by the lack of learning from experience that is evident in the industry but I am not sure that it is entirely down to the practitioners. I am beginning to think there is a structural problem with IT that exasperates people's tendency to want to reinvent everything and I think that it contributes and fuels the 'follow the hype' climate.

I think that the problem lies in the invisibility of software design in the delivered product. I think that this invisibility has at least two important consequences:

Firstly, it means that good design is not recognised by the users because they can not see it. There is no end-user pay-back for elegant design. Contrast this with civil engineering where design is very visible e.g. it is simple to see that the Millennium Bridge has a wonderfully elegant design. This means that the industry as a whole does not place much emphasis on quality of design, so practitioners do not see it as important either. Therefore if it is not important why learn about how others have done it in the past?

Secondly, it means that it is difficult for those that do want to learn to know where to start. You can gain a great deal of information about civil engineering and structural design by looking at buildings, bridges, etc. In fact the conversation of many architects will be peppered with references to this building or that bridge. Things are different for software engineers, there might be brilliant examples of software design installed on the computer I am using right now and I would never know. Even worse than not knowing, even if I did know I could not look at them because the source code will be secret.

This issue of secrets constantly nags at me. I think that the software industry's obsession with intellectual property is an important reason for the glacial pace of its advance. As an industry we keep things private and secret more than any other (at least in my experience). How is a rookie programmer supposed to learn how those that went before him did things if everything they produced is hidden behind a cloak of secrecy? Unless he is lucky enough to work in a very experienced team he is left blind. This point is made clear in your editorial: unless I am lucky enough to know your friends how could I know that they had solved the problems of the transactional programming model 25 years ago? It is easy to see how medieval architects solved the problems of load on cathedral walls, you can go and look at the flying buttresses!

One light on the horizon is the increasing use of Open Source Licensing. This has led of a large body of software being made available for those that want to learn. I for one have found that I have learnt more about software design from my involvement in a number of Open Source projects in the last few years than I have in most of my professional programming jobs. I have also found that I am more motivated to do an elegant design and professional coding job if I think that many people are likely to look at my code. I realise, that as a professional, I should always be motivated to do this, but I, like most practitioners in our industry, am only human.

So my advice to a new programmer that wants to "stand on the shoulders of giants" rather than be condemned to "reinvent the wheel" is this: Ignore just about everything the software vendors tell you, listen to the old hands on your team and spend some of your spare time reading and contributing to Open Source Software.

Regards

Richard Taylor

`rjt-accu@thegrindstone.me.uk`

Software's No Different...

Mark,

In many ways I would not classify software as being any different from many other industries. There is plenty of wheel reinventing going on all over the place (hence the creation of a term to describe it). Until we have lots of people regularly being killed for software related reasons, I don't see things changing.

This paper does an excellent job of covering the issues: `sunnyday.mit.edu/steam.pdf`

Derek M Jones

`derek@knosof.co.uk`

The Tale of a Struggling Template Programmer

by Stefan Heinzmann

By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race. *Alfred N. Whitehead*

I'm not exactly a beginner in C++ templates. Hey, I've even got "the book" on it [1]! But, admittedly, I'm not a guru – if I were, I wouldn't need the book. So I was fairly confident that I could easily come up with a little tool I have wanted to write for a while now. But it wasn't as easy as I thought. For your amusement I'm going to show you what I set out to do and what roadblocks I bumped into.

First, let me describe what I wanted to do. I have repeatedly come across the need to do table lookups in constant tables of key/value pairs. I program for embedded systems, so I am keen on keeping the table itself in nonvolatile memory (ROM, Flash, or whatever). I want to use binary search instead of linear search for efficiency. This requires the table to be stored in sorted order. Sorted according to the key alone, that is. Here's an example of such a table (binary search is not really beneficial with such a small table, but this is only a toy example):

```
const struct { int key; const char *value; }
table[] = {
  { 0, "Ok" },
  { 6, "Minor glitch in self-destruction module" },
  { 13, "Error logging printer out of paper" },
  { 101, "Emergency cooling system inoperable" },
  { 2349, "Dangerous substances released" },
  { 32767, "Game over, you lost" }
};
```

The first problem is to ensure the table is sorted. This is a compile-time task, so in theory it could be a job for template (or preprocessor) metaprogramming. If you're good at that, I've got some homework for you. It's too difficult for me. I just resolved to tell the programmer to make sure the table is sorted.

The second problem is simpler: Provide a lookup function that, given a key, returns the associated value, or a default value if the key could not be found in the table. Naturally, I wanted to implement this as a function template that adapts to the actual key and value types. As I want the tables to be put in nonvolatile memory, they will in practice be of POD type (C-style arrays of C-style structs). This is a much easier problem to solve, wouldn't you agree? So pour yourself a glass of your favorite beverage, install yourself in your armchair and chuckle while watching me making a fool of myself.

The STL contains binary search algorithms, so it is sensible to use them instead of inventing my own. The first attempt at my function template uses `std::equal_range` and goes like this:

```
template<typename Key, typename Val, unsigned n>
const Val &lookup(const std::pair<Key,Val>(&tbl)[n],
                  const Key &key, const Val &def) {
  typedef const std::pair<Key,Val> Entry;
  Entry entry(key, Val());
  std::pair<Entry*,Entry*> range
       = std::equal_range(tbl, tbl+n, entry);
  if(range.first != range.second)
    return range.first->second;
  return def;
}
```

You probably know or guessed that `std::equal_range` returns the range of entries which compare equal to the value to be searched. If there are no duplicates in the collection, this range will encompass either zero or one element. Also note the declaration of the function parameter `tbl`, which looks a bit weird at first glance. This declaration specifies that `tbl` is a reference to an array whose elements are of type `const std::pair<Key,Val>` and whose size is `n` elements. Since `Key`, `Val` and `n` are template parameters the compiler deduces all this information at compile time. This version of lookup hence requires the table to be an array of `std::pair` objects. The example table above thus has to be changed to use `std::pair<int,const char*>` instead of the anonymous `struct`. You use it like this:

```
std::cout << lookup(table,6,"???") << std::endl;
```

But alas, it doesn't work. As `std::pair` has constructors, it is not an aggregate, and thus can not be initialized with the curly braces notation. Try it: Your compiler will complain. For the same reason it will most probably not be put into read-only storage. Clearly, `std::pair` needs to be replaced by something that allows aggregate initialization:

```
template<typename Key, typename Val>
struct Pair {
  Key key;
  Val val;
};

template<typename Key, typename Val, unsigned n>
const Val &lookup(const Pair<Key,Val>(&tbl)[n],
                  const Key &key, const Val &def) {
  typedef const Pair<Key,Val> Entry;
  Entry entry = { key, Val() };
  std::pair<Entry*,Entry*> range
       = std::equal_range(tbl, tbl+n, entry);
  if(range.first != range.second)
    return range.first->val;
  return def;
}
```

That didn't work either. Here's what Visual C++ 7.1 had to say:

```
main.cpp(33) : error C2782: 'const Val
&lookup(const Pair<Key,Val> (&)[n],const Key
&,const Val &)' : template parameter 'Val' is
ambiguous
   main.cpp(11) : see declaration of 'lookup'
   could be 'const char [4]'
   or       'const char *'
```

Ok, fair enough, I thought, the compiler might have a point here. It can't figure out the proper type for `Val`, because it occurs twice in `lookup`'s signature (once as part of the `Pair`, once as the type of the `def` argument). What can we do about this? Well, I decided to try a separate template parameter for the array element type, like this:

```
template<typename Key, typename Val,
         typename Elem, unsigned n>
const Val &lookup(Elem(&tbl)[n],
                  const Key &key, const Val &def) {
  Elem entry = { key, Val() };
  std::pair<Elem*,Elem*> range
        = std::equal_range(tbl, tbl+n, entry);
  if(range.first != range.second)
    return range.first->val;
  return def;
}
```

I imagined that it would be easier for the compiler to figure out the various type promotions/decays when doing the initialization of the variable entry inside the function template. This stuff would then not get in the way when it tried to deduce the template parameters. The compiler still had something to complain about, however:

```
main.cpp(13) : error C2440: 'type cast' : cannot
convert from 'int' to 'const char [4]'
   There are no conversions to array types,
although there are conversions to references or
pointers to arrays
   main.cpp(32) : see reference to function
template instantiation 'Val (&lookup<int,const
char[4],const Pair<Key,const char *>,7>(Elem
(&)[7],const Key &,Val (&)))' being compiled
   with
   [
      Val=const char [4],
      Key=int,
      Elem=const Pair<int,const char *>
   ]
main.cpp(16) : error C2440: 'return' : cannot
convert from 'const char *const ' to 'const char
(&)[4]'
   Reason: cannot convert from 'const char *const '
to 'const char [4]'
   There are no conversions to array types,
although there are conversions to references or
pointers to arrays
```

The first error looked like complete idiocy to me. Why would the compiler want to convert an `int` to a `const char [4]` anyway? It had deduced the template arguments correctly, hadn't it? Compiler confusion!

Let's look at the second error then, as it actually made some sense to me. I'm using the `Val` template parameter for the return type as well as the default value, so it is no surprise that the compiler thinks it should be an array when I'm providing an array in the call to lookup. That can be fixed by forcing the pointer decay in the call to lookup, like this:

```
std::cout << lookup(table,6,(const char*)"???")
          << std::endl;
```

Disgusting, isn't it? It could be done with a more "politically correct" type of cast, of course, but it is actually a nuisance to need one in the first place. Let's see what we can do about this:

```
template<typename EKey, typename EVal,
         unsigned n, typename Key, typename Val>
const EVal &lookup(const Pair<EKey,EVal>(&tbl)[n],
                   const Key &key, const Val &def) {
  typedef const Pair<EKey,EVal> Elem;
  Elem entry = { key, Val() };
  std::pair<Elem*,Elem*> range
        = std::equal_range(tbl, tbl+n, entry);
  if(range.first != range.second)
    return range.first->val;
  return def;
}
```

Note that I reordered the template parameters to match the order of occurrence in the function argument list. I thought that was a good idea as the number of template parameters is growing. My stupid compiler still doesn't get it:

```
main.cpp(14) : error C2440: 'type cast' : cannot
convert from 'int' to 'const char [4]'
   There are no conversions to array types,
although there are conversions to references or
pointers to arrays
main.cpp(33) : see reference to function template
instantiation 'const EVal &lookup<int,const
char*,7,int,const char[4]>(const Pair<Key,Val>
(&)[7],const Key &,const char (&)) ' being compiled
   with
   [
      EVal=const char *,
      Key=int,
      Val=const char *
   ]
main.cpp(18) : warning C4172: returning address of
local variable or temporary
```

I still can't figure out why the compiler wants to convert an `int` to a `const char [4]`, but the warning is sensible: `Val` and `EVal` aren't necessarily the same, so the compiler is probably compelled to introduce a temporary in the last return statement. This temporary of course goes away too soon. Ok, that's easy to fix:

```
template<typename EKey, typename EVal,
         unsigned n, typename Key, typename Val>
EVal lookup(const Pair<EKey,EVal>(&tbl)[n],
            const Key &key, const Val &def) {
  typedef const Pair<EKey,EVal> Elem;
  Elem entry = { key, Val() };
  std::pair<Elem*,Elem*> range
        = std::equal_range(tbl, tbl+n, entry);
  if(range.first != range.second)
    return range.first->val;
  return def;
}
```

When `Val` is a `const char*` the return by value is no more expensive that the return by reference I used before. Alas, the value type could well be something more complicated (a `struct` full of stuff), so I'm not very happy to have a copy made. But if it must be it shall be...

The weird error remains, however. So how about asking a different compiler? Here's what GCC 3.3 says:

```
main.cpp: In function 'EVal lookup(const Pair<EKey,
EVal> (&)[n], const Key&, const Val&) [with EKey =
int, EVal = const char*, unsigned int n = 7, Key =
int, Val = char[4]]':
main.cpp:33:   instantiated from here
main.cpp:14: error: ISO C++ forbids casting to an
array type 'char[4]'
```

A number of further errors follow, but they're related to a different problem to which I'm going to turn later. Let's first try to find out what the error above means. Line 14 is where entry gets initialized. So what's wrong with this? When two compilers more or less agree, they might actually be right.

Both compilers seem to believe that I want a conversion from `EVal` to `Val` (give or take a `const`). I would have thought it ought to be the other way round. I construct a temporary of type `Val` and expect the compiler to convert it to an `EVal` in order to initialize the second field of the variable entry. In my particular case `Val` is an array and `EVal` is a pointer, so the conversion should simply be a decay. But why convert anyway? I've got an idea: Let's construct an `EVal` directly:

```
template<typename EKey, typename EVal,
         unsigned n, typename Key, typename Val>

EVal lookup(const Pair<EKey,EVal>(&tbl)[n],
            const Key &key, const Val &def) {
  typedef const Pair<EKey,EVal> Elem;
  Elem entry = { key, EVal() };
  std::pair<Elem*,Elem*> range
         = std::equal_range(tbl, tbl+n, entry);
  if(range.first != range.second)
    return range.first->val;
  return def;
}
```

That sorts it out. But it feels as if I had merely dodged the issue. I still don't know why the original attempt failed. If you know, tell me.

That brings us to the next errors I mentioned above. GCC emits a lot of error messages which are too numerous to print here, but most of them contain the text "`no match for operator<`", which is actually quite right since I obviously forgot to define how to compare a `Pair` to another. So I guess I have two options here. I could either provide `operator<` for a `Pair` or I could provide a custom predicate to `std::equal_range` to do the comparison. I only want to compare the keys, so I feel that the first option amounts to cheating. If someone else wanted to compare two `Pair`s and did not know about my ruminations here she would probably expect `operator<` to take both fields of the `Pair` into account.

So I think I should rather provide a special predicate with a sensible name like `LessKey` that makes it clear that only the key is being compared. Here we go:

```
template<typename Key, typename Val>
struct LessKey : std::binary_function<
                       const Pair<Key,Val>&,
                       const Pair<Key,Val>&,bool> {
  result_type operator()(first_argument_type a,
                        second_argument_type b) const
    { return a.key < b.key; }
};
```

Look Ma! I even made it adaptable by deriving from `std::binary_function`! I earned extra brownie points for that, didn't I? My lookup function template now looks like this:

```
template<typename EKey, typename EVal,
         unsigned n, typename Key, typename Val>
EVal lookup(const Pair<EKey,EVal>(&tbl)[n],
            const Key &key, const Val &def) {
  typedef LessKey<EKey,EVal> Pred;
  typedef const Pair<EKey,EVal> Elem;
  Elem entry = { key, EVal() };
  std::pair<Elem*,Elem*> range
     = std::equal_range(tbl, tbl+n, entry, Pred());
  if(range.first != range.second)
    return range.first->val;
  return def;
}
```

Needless to say, GCC still isn't happy. It emits a series of warnings regarding the implicit usage of `typename`. Here's just one of them:

```
main.cpp:14: Warning: 'LessKey<Key,
Val>::result_type' is implicitly a typename
main.cpp:14: Warning: implicit typename is
deprecated, please see the documentation for
details
```

The same happens for `first_argument_type` and `second_argument_type`. I could ignore those as they're only warnings, but I want to get it right. Furthermore I think I know what's amiss: I need to use the keyword `typename` to make clear that they are types. Next try:

```
template<typename Key, typename Val>
struct LessKey : std::binary_function<
                       const Pair<Key,Val>&,
                       const Pair<Key,Val>&,bool> {
  typename result_type operator()(
        typename first_argument_type a,
        typename second_argument_type b) const
   { return a.key < b.key; }
};
```

Now, GCC defecates an even bigger heap of error messages onto me ('scuse my French!). I don't want to spare you the experience:

```
main.cpp:13: error: Fehler beim Parsen before
'operator'
```

```
/usr/include/c++/3.3/bits/stl_algo.h: In function
'std::pair<_ForwardIter, _ForwardIter>
std::equal_range(_ForwardIter, _ForwardIter, const
_Tp&, _Compare) [with _ForwardIter = const
Pair<int, const char*>*, _Tp = Pair<int, const
char*>, _Compare = LessKey<int, const char*>]':
main.cpp:22:   instantiated from 'EVal lookup(const
Pair<Key, Val> (&)[n], const Key&, const Val&)
[with EKey = int, EVal = const char*, unsigned int
n = 7, Key = int, Val = char[4]]'
main.cpp:40:   instantiated from here
/usr/include/c++/3.3/bits/stl_algo.h:3026: error:
no match for call to '(LessKey<int, const char*>)
(const Pair<int, const char*>&, const Pair<int,
const char*>&)'
/usr/include/c++/3.3/bits/stl_algo.h:3031: error:
no match for call to '(LessKey<int, const char*>)
(const Pair<int, const char*>&, const Pair<int,
const char*>&)'
/usr/include/c++/3.3/bits/stl_algo.h: In function
'_ForwardIter std::lower_bound(_ForwardIter,
_ForwardIter, const _Tp&, _Compare) [with
_ForwardIter = const Pair<int, const char*>*, _Tp =
Pair<int, const char*>, _Compare = LessKey<int,
const char*>]':
/usr/include/c++/3.3/bits/stl_algo.h:3034:
instantiated from 'std::pair<_ForwardIter,
_ForwardIter> std::equal_range(_ForwardIter,
_ForwardIter, const _Tp&, _Compare) [with
_ForwardIter = const Pair<int, const char*>*, _Tp =
Pair<int, const char*>, _Compare = LessKey<int,
const char*>]'
main.cpp:22:   instantiated from 'EVal lookup(const
Pair<Key, Val> (&)[n], const Key&, const Val&)
[with EKey = int, EVal = const char*, unsigned int
n = 7, Key = int, Val = char[4]]'
main.cpp:40:   instantiated from here
/usr/include/c++/3.3/bits/stl_algo.h:2838: error:
no match for call to '(LessKey<int, const char*>)
(const Pair<int, const char*>&, const Pair<int,
const char*>&)'
/usr/include/c++/3.3/bits/stl_algo.h: In function
'_ForwardIter std::upper_bound(_ForwardIter,
_ForwardIter, const _Tp&, _Compare) [with
_ForwardIter = const Pair<int, const char*>*, _Tp =
Pair<int, const char*>, _Compare = LessKey<int,
const char*>]':
/usr/include/c++/3.3/bits/stl_algo.h:3036:
instantiated from `std::pair<_ForwardIter,
_ForwardIter> std::equal_range(_ForwardIter,
_ForwardIter, const _Tp&, _Compare) [with
_ForwardIter = const Pair<int, const char*>*, _Tp =
Pair<int, const char*>, _Compare = LessKey<int,
const char*>]'
main.cpp:22:   instantiated from 'EVal lookup(const
Pair<Key, Val> (&)[n], const Key&, const Val&)
[with EKey = int, EVal = const char*, unsigned int
n = 7, Key = int, Val = char[4]]'
main.cpp:40:   instantiated from here
/usr/include/c++/3.3/bits/stl_algo.h:2923: error:
no match for call to '(LessKey<int, const char*>)
(const Pair<int, const char*>&, const Pair<int,
const char*>&)'
```

I hope the occasional German word in there doesn't irritate you. Having part of the output of tools translated to German with the remainder in English leads to interesting effects. That's a story for another day.

Are you actually able to spot what's wrong from the gobbledygook above? I wasn't. Give me a hint if you are. All it tells me is that GCC ran into a parsing error in a system library because of the code I wrote (!). Is this a GCC bug? Visual C++ 7.1 is happy with it, no matter whether the `typename` keywords are there or not. So do I need them or not? How do I write this correctly?

Time to consult "the book"[1]! On page 131 we can find the following description:

"The `typename` prefix to a name is required when the name

1. Appears in a template
2. Is qualified
3. Is not used as in a list of base class specifications or in a list of member initializers introducing a constructor definition
4. Is dependent on a template parameter

Furthermore, the `typename` prefix is not allowed unless at least the first three previous conditions hold."

Ok, let's see whether I can make any sense of this: Rules 1 and 3 are obviously met. Rule 4 seems to be met indirectly, since `result_type` is defined by the base class `std::binary_function`, which in turn is a class template that depends on both template parameters `Key` and `Var`. However, rule 2 seems to be violated, as I can not see any qualification. So I conclude that `typename` shouldn't even be allowed where I put them. So GCC is technically right although it should have generated better error messages. Visual C++ accepted wrong code without complaint. But what is wrong with the original version without `typename`, why is GCC issuing a warning? Rule 2 pretty much excludes that a `typename` is missing elsewhere.

I solved the problem by not using `return_type` and its siblings from `std::binary_function`. The following is accepted by both GCC and Visual C++:

```
template<typename Key, typename Val>
struct LessKey : std::binary_function<
                     const Pair<Key,Val>&,
                     const Pair<Key,Val>&,bool> {
  bool operator()(const Pair<Key,Val> &a,
                  const Pair<Key,Val> &b) const
    { return a.key < b.key; }
};
```

Again, I had chickened out instead of solving the problem. Swallowing my pride, I went on to the next challenge:

```
const char *result = lookup(table,6,0);
```

I guess you see what I want to achieve?!? I want lookup to return a null pointer if the key wasn't found in the table. For a table that contains strings as values this is sensible, wouldn't you agree?

It doesn't compile. You already guessed why: The third argument to the lookup function gets interpreted as an integer as opposed to a null pointer, so the compiler comes up with the wrong choices for the template parameters and as a consequence the statement "`return def;`" fails to compile. I would have to write something like this:

```
const char *result = lookup(table,6,(char*)0);
```

But that's ugly; can you really expect that from an innocent user of my templates? What about this:

```
const char *result = lookup(table,6,NULL);
```

Nope, the compiler still interprets the NULL as an integer constant. Back to square 1. GCC's error message is actually quite interesting:

```
main.cpp: In function 'int main()':
main.cpp:40: Warnung: passing NULL used for non-
pointer converting 3 of 'EVal lookup(const
Pair<Key, Val> (&)[n], const Key&, const Val&)
[with EKey = int, EVal = const char*, unsigned int
n = 7, Key = int, Val = int]'
main.cpp: In function 'EVal lookup(const Pair<Key,
Val> (&)[n], const Key&, const Val&) [with EKey =
int, EVal = const char*, unsigned int n = 7, Key =
int, Val = int]':
main.cpp:40:   instantiated from here
main.cpp:25: error: invalid conversion from 'const
int' to 'const char*'
```

The compiler does indeed notice that `NULL` is being used as a non-pointer, but it still pigheadedly decides to instantiate the template with `Val = int`.

Despair sets in, creeping slowly from the back of my head towards the front. Is there any hope to get this right? Am I trying to do something frivolous? Illegal? Immoral? Fattening?

Time to step back and take a deep breath. What can we learn from this?

It doesn't take much to get into serious trouble with templates. This doesn't mean that templates themselves are to blame. It is rather the interaction between templates and other C++ "features" that cause problems. It is similar to combining drugs: Each one is harmless when taken alone, but taken together they might kill you.

Over the last years we have seen an increasing number of tricks and workarounds attempting to control and contain the unwanted effects of these interactions. Library designers are forced to know and use them to prevent unpleasant surprises for the mere mortal library user. Here's a short list of idiosyncrasies that I just came up with ad-hoc. You're invited to add your favorites to it. Maybe we could create a "C++ derision web page" from it. I hope you don't take this seriously enough to be offended.

- Angle brackets being mistaken as shift operators in nested template declarations
- The need to put in additional `typename` keywords in obscure circumstances
- The need to put in additional `template` keywords in even more obscure circumstances
- The need to use `this->` explicitly to control name lookup in templates in another set of obscure circumstances
- The often unwanted effects of automatic type conversions/promotion/decay on template argument deduction and overload resolution.
- The fact that `bool` converts to `int` automatically. This has the effect that returning a member function pointer is sometimes superior to returning a `bool` because it doesn't convert as easily. (Savour this: Member function pointers are better booleans than `bool`!)
- The fact that the literal `0` converts to the null pointer.
- The C/C++ declaration syntax (need I say more?)
- Argument-Dependent Lookup

To me it seems the more experience I've gained programming in C++ the more I realize how inadequate my skill still is. I have been programming in C++ for a considerable time now, I have read numerous books on the subject, and still I don't seem to have it under control. This bothers me. How can an average programmer be expected to cope with this? Imagine for a moment yourself giving a lecture to a classroom full of the type of programmers you meet in your working life. Your topic is the sort of stuff I came across in my example above. Just imagine the puzzled faces.

I don't know what to do about this. If the other languages weren't even worse in one way or another I would probably switch. But so far I haven't seen anything I like 100%. I had a look at Haskell, and I generally liked what I saw, particularly regarding generic programming, but I couldn't see how to apply this to embedded programming, where you deal extensively with low-level stuff that is rather close to the hardware level. Here I would like to be able to predict what kind of code is being generated. With Haskell I feel I have no control over this. Maybe that's because I'm not experienced enough with it.

My dream language would take Haskell's elegant type inference machinery and built-in list/tuple/array manipulation and combine it with a more "conventional" syntax. It must support object-oriented and imperative programming styles, because I can't see how I could do without them in embedded programming.

My feeling is that languages that have genericity tagged on as an afterthought are not really satisfactory. Backwards compatibility is likely to prevent cleaning up the rough edges. The result is the sort of thing we have now with C++: The language is very complicated and has lots of obscure and surprising special cases. I am asking myself if it would not be possible to deprecate troublesome features in a language more aggressively. I know of course that C++ has become popular precisely because of its backward compatibility and that right now the standard committees for C and C++ are cooperating closely to ensure that this remains so.

Back in now ancient times the ANSI C standard introduced function prototypes, offering a better alternative to K&R style function declarations. I haven't seen any K&R style code for a while now. It still exists, but it is now customary to require a compiler switch to make the compiler accept the old form. Can't something similar be done with C++? Something like a C++ generation 2 that is not backwards compatible with the current generation, but is link-compatible with it. And the compiler would continue to accept the old version while it is gradually being phased out.

To wind up, let's see where we are with my little problem. Here's the complete code as it stands now:

[concluded at foot of next page]

Lvalues and Rvalues

by Mikael Kilpeläinen

The two concepts, *lvalue* and *rvalue*, can be somewhat confusing in C++. Nevertheless, the difference is important to understand. The basic consequences related to these concepts are interesting and good to know in many cases.

In C++ every expression yields either an lvalue or an rvalue and accordingly every expression is called either an lvalue or an rvalue expression. An example of an lvalue is an identifier. As a further example, a reference to an object is an lvalue. Every expression that is not an lvalue is an rvalue. A good example of this is an expression that produces an arithmetic value. An intuitive approach would be to think of expressions as functions and then an lvalue can be thought as the result of a function returning a reference.

Examples:

1. The subscript operator is a function of the form `T& operator[](T*, ptrdiff_t)` and therefore `A[0]` is an lvalue where `A` is of array type.
2. The dereference operator is a function of the form `T& operator*(T*)` and hence `*p` is an lvalue where `p` is of pointer type.
3. The negate operator is of the form `T operator-(T)` and therefore `-x` is an rvalue.

The terms lvalue and rvalue were inherited from C. The original meaning comes from the assignment: an lvalue being the left side of the assignment and rvalue the right side. However, in the modern C++, lvalue can be considered more as a *l*ocator *value*. An lvalue refers to a defined region of storage. Although, this is not true with the function lvalues since functions are not objects. Similarly, an rvalue can be considered as a value of an expression. This separation of two concepts helps to define and talk about things, though some would say that it has caused more confusion. Nevertheless, exact definitions are needed to clarify a language.

An rvalue should not be confused with the constness of an object. An rvalue does not mean the object would be immutable. There is some confusion about this, since non-class rvalues are non-modifiable. This is not the case with user types. A class rvalue can be used to modify an object through its member functions. Albeit in practice, it can be said that objects are modified only through *modifiable lvalues*. A modifiable lvalue is an lvalue that can be used to modify the object. Other lvalues are non-modifiable lvalues, const reference is a good example of this.

[continued from previous page]

```
#include <iostream>
#include <algorithm>
#include <functional>

template<typename Key, typename Val>
struct Pair { Key key; Val val; };

template<typename Key, typename Val>
struct LessKey : std::binary_function<
    const Pair<Key,Val>&, const Pair<Key,Val>&,bool> {
  bool operator()(const Pair<Key,Val> &a,
                  const Pair<Key,Val> &b) const
    { return a.key < b.key; }
};

template<typename EKey, typename EVal,
         unsigned n, typename Key, typename Val>
EVal lookup(const Pair<EKey,EVal>(&tbl)[n],
            const Key &key, const Val &def) {
  typedef LessKey<EKey,EVal> Pred;
  typedef const Pair<EKey,EVal> Elem;
  Elem entry = { key, EVal() };
  std::pair<Elem*,Elem*> range
     = std::equal_range(tbl, tbl+n, entry, Pred());
  if(range.first != range.second)
    return range.first->val;
  return def;
}
const Pair<int, const char*> table[] = {
  { 0, "Ok" },
  { 6, "Minor glitch in self-destruction module" },
  { 13, "Error logging printer out of paper" },
  { 101, "Emergency cooling system inoperable" },
  { 2349, "Dangerous substances released" },
  { 32767, "Game over, you lost" }
};

int main() {
  const char *result = lookup(table,6,(char*)0);
  std::cout << (result ? result : "not found")
            << std::endl;      }
```

It has the following problems:

- It requires the ugly cast for passing the null pointer as the third argument to `lookup`
- `lookup` returns the result by value, which can be inefficient
- It is still unclear why I couldn't use the typedefs from `std::binary_function` in the `LessKey` predicate (only with gcc)
- Neither do I know why the compiler wanted to convert the wrong way between `Val` and `EVal`

So if you want to have a go, you're invited to contribute your solutions. I'd like to know how this is done right!

Stefan Heinzmann
stefan_heinzmann@yahoo.com

Conventional Programming Languages: Fat and Flabby

For twenty years programming languages have been steadily progressing toward their present condition of obesity; ... it is now the province of those who prefer to work with thick compendia of details rather than wrestle with new ideas. *John Backus, 1977*

Acknowledgements

Thanks to the reviewers Phil Bass, Thaddaeus Frogley and Alan Griffiths for their comments.

The Book

[1] D. Vandevoorde, N. M. Josuttis, *C++ Templates: The complete guide*, Addison-Wesley, 2003

Developers are people too, and hate to admit that they are confused or that their understanding is incomplete. We should be grateful therefore that Stefan was willing to write this article which illustrates the potential C++ has for causing these symptoms even in experienced developers. It is said that an admission of ignorance is the first step to wisdom and further steps were taken when one of Overload's Readers became interested in solving the problem presented above. If you wish to tackle this exercise yourself then you may want to postpone reading the results of their collaboration (which, in the traditional manner, appears towards the end of this magazine). (AG)

As mentioned already, non-class rvalues do not have the same qualities as the user type rvalues. One might wonder about this. After all, C++ was designed so that user types would behave like built-ins, at least as uniformly as possible. Still this inconsistency exists and the reasons for it shall be explored later. Non-class rvalues are not modifiable, nor can have cv-qualified types (the cv-qualifications are ignored). On the contrary, the class rvalues are modifiable and can be used to modify an object via its member functions. They can also have cv-qualified types. In case of built-ins, some operators require an lvalue as does every assignment expression as the left side. The built-in address-of operator requires an lvalue which reflects the character of lvalues rather well.

Examples:

```
int var = 0;
var = 1 + 2; // ok, var is an lvalue here
var + 1 = 2 + 3; // error, var + 1 is an rvalue
int* p1 = &var; // ok, var is an lvalue
int* p2 = &(var + 1); // error, var + 1 is an rvalue
UserType().member_function(); // ok, calling a
            //  member function of the class rvalue
```

The only real reason I can think of for needing the class rvalues to be modifiable, is to allow the calling of the non-const members of the proxies[1] and similar. That is, a proxy represents a type and it ought to behave accordingly. Keeping this in mind when considering coherent behaviour along with the const-correctness, one would make the mutating members non-const. For this to work the modifiable rvalues are needed. Although this may seem quite irrelevant, it is actually quite an important reason. It can be used to emulate lvalue behaviour and hence many things are made possible. Thinking about the modifiable class rvalues more closely, they enable many usable concepts and there are only a few rare cases when that introduces problems. After all, the big difference between the built-in types and the user types is that the user types can have members. This difference effectively makes the non-class rvalues non-modifiable.

An rvalue cannot be used to initialise non-const reference. That is, an rvalue cannot be converted to an lvalue, but when an lvalue is used in a context where an rvalue is expected, the lvalue is implicitly converted to an rvalue. This binding restriction and the modifiable class rvalue lead to interesting consequences. It allows us to call all member functions for a user type but not mutating free functions. This can be confusing as one needs to know whether an operator is a member or not, and after all it is not consistent. The same problem motivates us to implement operators as non-members where possible, for consistency with built-in types. Also, since the member functions can be called, the called function can return a non-const reference to the object itself. This means that a modifiable lvalue referring to a temporary object can be created, making it possible to call a function that takes a non-const reference. Time has shown this to be very dangerous as it allows mistakes that can be hard to find, mostly because of the implicit conversions. That ought not to be the case here, since it is not easy to make such a mistake by accident.

Examples:

```
struct A {
  A& operator=(const A&) { return *this; }
};
void func(A&);
..
func(A() = A()); // fine, operator= yields an lvalue

ofstream(“some”) << some_variable; // fine as long
                     // as the operator<< is a member
```

Forbidding the binding of rvalues to non-const references does not come without problems, however. It makes it difficult to provide uniform behaviour with unusual copy semantics, like those of `auto_ptr`, which is why they are a bad idea. This is why a special reference has been proposed for the next C++ standard. It would only bind to an rvalue. The proposal actually makes a distinction between the rvalue and the lvalue references by introducing a new syntax. This would allow detection of an rvalue which is crucial for the move semantics[2]. The proposed resolution would effectively allow the uniform behaviour but still not compromise on the safety issues.

The biggest problem with non-consistency seems to be the confusion among the people. Of course it would be desirable to be consistent in the eyes of purists but you cannot always get it all. After taking a little trouble to understand, an lvalue and an rvalue are quite easy concepts.

Mikael Kilpeläinen
mikael.kilpelainen@kolumbus.fi

Acknowledgement

I would like to thank Rani Sharoni for providing helpful comments.

References

[1] ISO/IEC 14882-1998 Standard for the C++ language
[2] ISO/IEC 9899-1999 Programming language C
[3] Andrew Koenig and Barbara E. Moo, *Accelerated C++, Practical Programming by Example*, 2000, Addison Wesley
[4] Bjarne Stroustrup, *The Design and Evolution of C++*, 1994, Addison Wesley
[5] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1995, Addison Wesley
[6] Howard E. Hinnant, Peter Dimov and Dave Abrahams, *A Proposal to Add Move Semantics Support to the C++ Language*, 2002, `http://std.dkuug.dk/jtc1/sc22/wg21/docs/papers/2002/n1377.htm`

Related links

[1] `auto_ptr` update, Scott Meyers, `http://www.awprofessional.com/content/images/020163371X/autoptrupdate%5Cauto_ptr_update.html`
[2] Generic<Programming>: Move Constructors, Andrei Alexandrescu, `http://www.cuj.com/documents/s=8246/cujcexp2102alexandr/alexandr.htm`

[1] Proxy: provide a surrogate or placeholder for another object to control access to it.

[2] A proposal to add support for move semantics to the C++ has been made and the rvalue-reference proposal is part of it.

When is a “Pattern” not a “Pattern”?

by Alan Griffiths

We all go through life listening to and telling stories: it is an important part of human social behaviour. Parents tell children bedtime stories, we read novels, watch TV and films, and many successful computer games are based on allowing the player to participate within a story. Incidentally, one of the things that that seems to have been lost in many of these formats is the idea of participation – although, to the annoyance of their mother, my children re-invented it for bedtime stories (she preferred reading from a book to collaborative invention).

Stories are also important in the development of computer software. Not only the stories about how the system will be used, but also the stories about how it will work. As with the stories in the wider world they can take a number of forms: Among these are “design patterns” and that is the form that I want to examine in this essay.

Before continuing, I should mention that there are those that think “Patterns” are like the Platonic Ideals to be found in philosophy: unchangeable and perfect examples of a concept. (This is not to say that any particular write-up of a pattern is ever perfect – Plato’s Ideals exist in the “real world”, not the mundane one.)

For me the important aspect of a pattern is that it narrates the way that a design context is transformed by the application of a solution. I don’t expect to ever meet an idealised design context and so don’t demand that a pattern is the one true solution to a design problem. (Only that it tells me enough to decide for myself if applying the solution will advance the task.)

To demonstrate this I am going to present two “patterns” with a common beginning, one that many developers are familiar with. In fact, a good number of developers will be familiar with the whole of both stories to the extent that I’m not going to worry about some of the formalities of writing about patterns – such as citing “existing implementations”. These are left as an exercise for the interested reader.

Story Number 1

Initial Context

When a program is working with numerical values it is necessary to represent these using the types and names available in the language. Many languages provide some native data types (such as the C/C++ types `int` and `double`) which support a rich array of operations and conversions. Some of these operations and conversion may not be appropriate to the types of numerical value being worked with. For example, `double` might provide the range of values and operations required for working with temperature and pressure. However, it is possible to make mistakes by, for example, assigning a temperature value to a pressure variable.

Problem

How can such (hard to diagnose and expensive to fix) mistakes be avoided?

Solution

Encode the problem domain types in the variable names so that the programmer is reminded of the appropriate use of the variable. For example, by using a prefix indicating the type of the variable. Vis: `cgdEngine` and `ntnsOil`.

Resulting Context

The developer can see from its name the appropriate manner in which to make use of a variable. An error like:

```
ntnsOil = cgdEngine
```

is noticeable “at a glance”. However, there is an overhead to this solution: the developer needs to maintain a catalogue of problem domain type prefixes and the corresponding types within the programming language. Note that this catalogue may be explicit (written down) or implicit (what everyone knows).

This is a *perfectly good pattern* that is adequately captured in the above form. It would gain little from written in a more formal structure (e.g. Coplien). It could also be written in a less formal manner (e.g. Alexandrian) and still be considered a valid pattern.

As for existing usage: this is one of several design approaches referred to as “Hungarian Notation” and there is no shortage of developers willing to attest to its effectiveness.

Story Number 2

Initial Context

When a program is working with numerical values it is necessary to represent these using the types and names available in the language. Many languages provide some native data types (such as the C/C++ types `int` and `double`) which support a rich array of operations and conversions. Some of these operations and conversion may not be appropriate to the types of numerical value being worked with. For example, `double` might provide the range of values and operations required for working with temperature and pressure. However, it is possible to make mistakes by, for example, assigning a temperature value to a pressure variable.

Problem

How can such (hard to diagnose and expensive to fix) mistakes be avoided?

Solution

Encode the problem domain types as user defined types so that the compiler enforces the appropriate use of the variable. For example, types “`centigrade`” and “`newtons`” may be defined so that they only support the appropriate operations and conversions.

Resulting Context

An error like:

```
oil_pressure = engine_temperature
```

will not compile. However, there is an overhead to this solution: the developer needs to maintain a library of problem domain types and code the support for the allowed operations and conversions.

Like the first story this is a *perfectly good pattern* but it is clearly a different one.

Once again it isn’t hard to find developers that are willing to attest to the effectiveness of this approach.

Two Paths to Choose From

What are we to make of this? We have two “patterns” that start from the same position, and apply different solutions with “success”. Which design should we choose for our work?

[concluded at foot of next page]

Efficient Exceptions?

by Roger Orr

I recently read a comment on a code review that produced a fair amount of discussion about exceptions. Slightly simplified, the C# code being reviewed was:

```
public bool isNumeric(string input) {
  bool ret = true;
  bool decimalFound = false;

  if(input == null
          || input.Length < 1) {
    ret = false;
  }
  else {
    for(int i = 0
        ; i < input.Length
        ; i ++) {
      if(!Char.IsNumber(input[i]))
        if((input[i] == ‘.’)
                 && !decimalFound) {
          decimalFound = true;
        }
        else {
          ret = false;
        }
    }
  return ret;
}
```

The review comment that started the discussion was quite short:

Your isNumeric function might be more efficient as:

```
public bool isNumeric(string input) {
  try {
    Double.Parse(input);
    return true;
  }
  catch(FormatException /*ex*/) {
    return false;
  }
}
```

I thought some of the discussion inspired by this comment might be of more general interest. There are several important issues that are relevant here, and later in the article I will return to some of them. However the first thing that struck me was the use of the word “efficient” and it is this word that the bulk of the article addresses.

What Does it Mean to be Efficient?

In the coding context efficiency is usually concerned with size and/or speed.

The second piece of code is more efficient in terms of source code size. And it is probably slightly more efficient in terms of image code size; but it is almost certainly not more efficient in terms of runtime memory use, particularly on failure, since exceptions in C# will allocate memory.

So I started wondering about runtime efficiency – which for simplicity I will from here on refer to as ‘performance’. Would the proposed replacement function be any faster than the original function?

[continued on next page]

[continued from previous page]

Well, it would be easy if one solution always leads to a better result than the other does. Then one pattern would be better than the other (and closer to that Ideal). But this turns out not to be the case – it is not always the same solution that produces the better result.

Clearly both approaches resolve the problem stated in the initial context. But when the two resulting contexts are examined closely it becomes apparent that a careful assessment of costs and benefits is required. Specifically: we have to chose between the cost of maintaining the “prefix catalogue” or of maintaining the “type library” described in the two solutions. (This may appear to be lower in the first story as the catalogue need not be maintained explicitly.) At the same time we have to choose between the benefits of visual and compiler based error detection.

In some circumstances the second solution may not even be an option: the cost of implementing user-defined types may be so high as to be prohibitive – in some languages there is no support for it. (Although it may be possible to add them – some “lint” style tools add the notion of “strong typedef” to C for this purpose.) Even where possible in the language, the developer skills available to the project may not be up to providing the desired user defined types reliably.

It is far rarer that the first solution isn’t available: languages whose maximum identifier length is significantly constrained are thankfully rare these days. But there are occasions where identifiers are displayed to users who would not appreciate these prefixes.

It could even be worse: the initial context may sometimes be better than either of the resulting contexts described above. This happens whenever the cost of introducing a new type is higher than the risk involved in reusing an existing type – or when the language doesn’t allow user-defined types or long identifiers.

Will the True Pattern Please Stand Up?

What is this: two (or three) patterns that might apply to the same problem? That doesn’t fit with the popular view of patterns as a template for solving problems: match the initial context and the problem, and the solution follows automatically.

One way to defend this viewpoint is to insist that there are additional forces at work, ones not captured by the initial context. But look at what these forces are: the cost and/or benefits of aspects the resulting context. Discussing these as part of the initial context would seriously unbalance the story and lead to turgid prose. No, there is no escaping from it: there is still a role for the designer! These patterns don’t replace thought – you still have to decide which option is better.

Alan Griffiths
alan@octopull.demon.co.uk

The Dangers of Performance Figures

Note: performance figures are very dangerous! They depend on all sorts of factors, such as the language being used, the compiler settings, the operating system being used and the hardware that the program runs on.

Although I’ve done my best to produce repeatable performance figures for this article please do not take any of the figures as being more than indicative of the overall performance of the languages mentioned. A small change to the programs being tested could produce variations in the figures produced.

For those who care I was using Windows 2000 SP4 on a 733 MHz single CPU machine with 768 Mb of RAM. (Yes, maybe it’s time I bought a newer machine!)

I was using:

- C# with csc from Visual Studio .NET (version 7.00.9466), both with and without optimising
- Java with JDK 1.3.0 and JDK 1.4.2
- C++ with Microsoft VC++ 6 (version 12.00.8804) both with (`/Ox`) and without (`/Od`) optimising. In both cases I was also using `/GX /GR`.
- C++ with gcc 2.95.3-4 under Cygwin (with and without `-O`)

(I also repeated a couple of the C++ tests with the Microsoft .NET and .NET 2003 C++ compilers but the results did not change enormously.)

It is important to note that I was *not* principally looking to optimise the hand written code – I was interested in the effect on performance of using exceptions. For this reason I deliberately kept the implementations of each function similar to ones in the other languages – hence, for example, the use of the member function `at()` in the C++ code rather than the more idiomatic `[]` notation. In fact, after being challenged about this, I tested both methods and to my shock found that `at()` was actually faster than `operator[]` with MSVC6. If you find this unbelievable it only goes to show how unexpected performance measurements can be, and how dependent they are on the optimiser!

I also made the `IsNumeric` method an instance method of a class in all languages for consistency and ease of testing. Changing this would have equally affected the performance of both the exception and the non-exception code so I left it as it was.

It is often said that it is better to use library functions than to write your own code; apart from any other considerations library functions are often optimised by experts using a wide variety of techniques. However, in this case using the library function adds exception handling into the equation – would the advice still stand?

I thought I’d try to get some actual performance figures.

I wrote a simple test harness that called the first and the second functions 1,000,000 times.

(execution times in seconds)

Unoptimised C#

Argument	**Function #1**	**Function #2**
1	0.19	0.92
12345	0.56	1.13
10 digits	1.01	1.38
20 digits	1.91	1.79

Optimised C#

Argument	**Function #1**	**Function #2**
1	0.10	0.89
12345	0.33	1.01
10 digits	0.65	1.36
20 digits	1.28	1.76
30 digits	2.07	1.92
40 digits	2.55	2.38

So the first function is quite a bit faster for relatively short strings, but degrades until it is eventually slower than the second function. Similar results are generated when optimisation is turned on, although the number of digits at the ‘break even’ point is slightly more.

The main question I was investigating though is what happens when a non-numeric value is supplied and an exception is thrown.

Argument	**Function #1**	**Function #2**	
X	0.20	147.60	(unoptimised)
X	0.11	143.24	(optimised)

Yes, that’s right – the decimal point is in the right place for function #2! The code path through the exception throwing route took almost 3 orders of magnitude longer than the raw code.

This is why, for this article, I’m just not interested in minor optimisations of the source code, since the impact of exceptions dwarfs them.

This was very intriguing – I wondered whether it was only a C# issue or it was also an issue with Java and C++.

Here is an approximately equivalent pair of functions in Java:

```
public boolean isNumeric(String input) {
  boolean ret = true;
  boolean decimalFound = false;

  if(input == null
         || input.length() < 1) {
    ret = false;
  }
  else {
    for(int i = 0
        ; i < input.length()
        ; i ++) {
      if(!Character.isDigit(
                        input.charAt(i)))
        if((input.charAt(i) == ‘.’)
                && !decimalFound) {
          decimalFound = true;
        }
        else {
          ret = false;
        }
    }
  }
  return ret;
}
```

and:

```
public boolean isNumeric(String input) {
  try {
    Double.parseDouble(input);
    return true;
  }
  catch(NumberFormatException ex) {
    return false;
  }
}
```

Surely code that looks so similar must behave the same way :-) ?

Here are the results for the non-exception case:

jdk 1.3.0

Argument	**Function #1**	**Function #2**
1	0.13	0.81
12345	0.42	1.15
10 digits	0.76	1.68
20 digits	1.48	23.16

jdk 1.4.2

Argument	**Function #1**	**Function #2**
1	0.10	0.76
12345	0.29	1.12
10 digits	0.51	1.63
20 digits	0.94	28.08

The results here are comparable to the optimised C# results – apart from the last line. What happens here? The `parseDouble` method is much slower once you exceed 15 digits – this is to do (at least with the versions of Java I'm using) with optimisations inside `Double.parseDouble` when the number is small enough to be represented as an integer value. Whether this matters in practice of course depends on the range of input values the program actually passes to the `isNumeric` function.

The exception results look like this:

Argument	**Function #1**	**Function #2**	
X	0.16	15.33	(jdk 1.3.0)
X	0.12	18.15	(jdk 1.4.2)

Well, this is not quite as awful as the C# case – the performance of the second function is ‘only’ two orders of magnitude worse than the first function when the exception is thrown.

For completeness, how about a C++ solution?

The roughly equivalent functions I came up with were:

```
bool IsNumeric1::isNumeric(std::string
                   const & input) const {
  bool ret = true;
  bool decimalFound = false;

  if(input.length() < 1) {
    ret = false;
  }
  else {
    for(int i = 0
        ; i < input.length()
        ; i ++) {
      if(!isdigit(input.at(i)))
        if((input.at(i) == ‘.’)
                && !decimalFound) {
          decimalFound = true;
        }
        else {
          ret = false;
        }
    }
  }
  return ret;
}
```

and:

```
bool IsNumeric2::isNumeric(std::string
                   const & input) const {
  try {
    convert<double>(input);
    return true;
  }
  catch(std::invalid_argument const & ex) {
    return false;
  }
}
```

where `convert` was derived from code in `boost::lexical_cast` from `www.boost.org` (in the absence of a standard C++ library function with similar syntax and semantics to the C# and Java `parse` functions) and looks like this:

```
template<typename Target>
Target convert(std::string const & arg) {
  std::stringstream interpreter;
  Target result;
  if(!(interpreter << arg)
      || !(interpreter >> result)
      || !(interpreter >> std::ws).eof())
    throw std::invalid_argument( arg );
  return result;
}
```

I decided using a reference in C++ kept the source code looking more equivalent although a smart pointer could have been used instead as its behaviour is more like that of a reference in the other two languages.

How did C++ fare in the comparison?

MSVC unoptimised

Argument	**Function #1**	**Function #2**
1	0.14	13.04
12345	0.46	17.46
10 digits	0.83	23.83
20 digits	1.57	34.31

MSVC optimised

Argument	Function #1	Function #2
1	0.07	5.73
12345	0.21	7.65
10 digits	0.40	11.25
20 digits	0.74	17.12

Our initial choice for the `convert` function is very slow – perhaps it is a bad choice. The cost of using `stringstream` objects seems to be very high, although that might be a problem with my compilers' implementations. This is not really an entirely fair comparison either – the `convert` template function is generic whereas the C# and Java code is type-specific. So let me replace the generic `convert` function with:

```
double convert(std::string const & arg) {
  const char *p = arg.c_str();
  char *pend = 0;
  double result = strtod(p, &pend);
  if(*pend != '\0')
    throw std::invalid_argument(arg);
  return result;
}
```

This produces the following improved performance figures:

MSVC unoptimised

Argument	Function #1	Function #2
1	0.14	1.82
12345	0.46	2.71
10 digits	0.83	5.10
20 digits	1.57	8.62

MSVC optimised

Argument	Function #1	Function #2
1	0.07	1.80
12345	0.21	2.71
10 digits	0.40	4.94
20 digits	0.74	8.39

And finally I recompiled the C++ code with gcc under Cygwin.

gcc unoptimised

Argument	Function #1	Function #2
1	0.29	0.63
12345	0.98	0.70
10 digits	1.79	0.85
20 digits	3.44	3.87

gcc optimised

Argument	Function #1	Function #2
1	0.05	0.33
12345	0.11	0.40
10 digits	0.17	0.55
20 digits	0.27	3.55

However, what about the exception throwing case (which is after all the motivating example) ?

Argument	Function #1	Function #2	
X	0.17	11.69	(MSVC unoptimised)
X	0.08	11.03	(MSVC optimised)
X	0.40	4.15	(gcc unoptimised)
X	0.06	3.17	(gcc optimised)

Even discounting the cost of solution #2 there is one to two orders of magnitude difference between the return code and exception throwing case, but with some significant differences between the two compilers.

What is the Cost of an Exception?

Exceptions tend to be expensive for a number of reasons, described below:

The exception object itself must be created.
This is not usually very expensive in C++, although it does obviously depend on the exact class being used. In Java and in C# the exception object contains a call stack, and the runtime environment has to create this before the exception is thrown. This may be quite expensive, particularly if the function call stack is deep.

The act of throwing the exception can be expensive.
For example, when using Microsoft compilers under Windows, throwing a C++ exception involves calling the OS kernel to raise an operating system exception, which includes capturing the state of the thread for passing to the exception handler. This approach is by no means universal – gcc under Cygwin does not use native operating system exceptions for its C++ exceptions, which seems to have as a consequence that the execution time cost of an exception is lower.

Then, in C++, a copy of the supplied object is thrown, which can impose some overhead for non-trivial exception objects.

There is the cost of catching the exception.
This in general involves unwinding the stack and finding suitable catch handlers, using run time type identification to match the types of the thrown object to each potential catch handler. For example, if I throw a `std::invalid_argument` object in C++ this might be caught by:

- `catch(std::invalid_argument const &)`
- `catch(std::exception)`
- `catch(...)`

with different behaviour in each case. The cost of this rises with both the depth of the exception class hierarchy and the number of catch statements that there are between the throw and the successful catch.

Note that some experts in compiler and library implementation claim that high performance exception handling is theoretically possible; however in practice it seems than many of the popular compilers out there do have less than optimal performance in this area.

Should I Care How Slow Exceptions Are?

Let's take stock of where we have reached. I've investigated the 'efficiency' claims for the proposed replacement code and found that it is almost always slower for numeric input and very significantly slower for non-numeric input.

On examining the two functions you can quickly see that they do not produce the same answers for all inputs; this is probably much more significant than which function runs faster since in most applications 'right' is better than 'fast but wrong'.

Consider the results the two C# implementations give for the following inputs:

Input	Function #1	Function #2
"+1"	False	True
"-1"	False	True
"."	True	False
" 1"	False	True
"1 "	False	True
"1e3"	False	True
"1,000"	False	True
"Infinity"	False	True
null	False	Exception

The library function understands a much broader range of numeric input than the hand-crafted code does. And that's leaving aside all discussion about locales (should ',' be a decimal point or a thousands separator?), which the library function takes in its stride. This probably provides an explanation of why our own conversion function is faster than the library call – it isn't a complete solution!

The problem with the initial code review was the word 'efficient'; I would like to make use of the library call to take advantage of its rich functionality despite the loss of efficiency. However I'd like to reduce the expense of the exception – is this possible?

The exception is being thrown when the input is not numeric so its cost only matters in this case. Ideally I'd like to find out how many times the function returns false in typical use; unfortunately a simple profiler will only tell me how many times the function is called and not differentiate on return code. I either need to use a better profiler or to add some instrumentation to my program.

In the best case I might find that the function usually succeeds and then I probably don't mind taking a performance hit on the rare failures. However I might find that the function is called a lot and is roughly evenly divided between success and failure – in this case I will want to reduce the cost.

As it happens, it is fairly easy to do this in the C# case. Closer investigation of the `Double` class reveals a `TryParse` method that has exactly the behaviour we require in `IsNumeric`. It needs a couple of additional arguments but the resultant code is clear:

```
using System.Globalization;
...
public bool isNumeric(string input) {
  double ignored;
  return = Double.TryParse(input,
         NumberStyles.Float |
              NumberStyles.AllowThousands,
         NumberFormatInfo.CurrentInfo,
         out ignored);
}
```

The results of running this function are:

Optimised C#

Argument	Function #1	Function #2	Function #3
1	0.10	0.89	1.08
12345	0.33	1.01	1.29
10 digits	0.65	1.36	1.55
20 digits	1.28	1.76	1.95
X	0.11	143.24	1.90

Unfortunately `Double.Parse(string)` is slightly faster than `TryParse` for the 'good' case but this is outweighed by the drastic improvement in speed on 'bad' inputs. In the absence of specific measurements of performance I would prefer this solution.

The Java case is more difficult – there is no direct equivalent to `TryParse`. I tried the following:

```
public boolean isNumeric(String input) {
  java.text.NumberFormat numberFormat
       = java.text.NumberFormat.
                           getInstance();
  java.text.ParsePosition parsePosition
       = new java.text.ParsePosition(0);

  Number value = numberFormat.parse(
                   input, parsePosition);
  return ((value != null)
          && (parsePosition.getIndex()
                  == input.length()));
}
```

However the performance is 'disappointing'. The new method is indeed faster when an exception occurs – but an order of magnitude slower when the input is in fact numeric. The biggest cost is creating the `numberFormat` object – caching this makes it a lot faster, but additional coding work would need to be done to make it threadsafe (see the JDK 1.4 documentation for `NumberFormat`).

jdk 1.3.0

Argument	Funct'n #1	Funct'n #2	Funct'n #3	Funct'n #3 + caching
1	0.13	0.81	14.54	2.39
12345	0.42	1.15	16.00	3.77
10 digits	0.76	1.68	18.11	5.88
20 digits	1.48	23.16	51.19	34.70
X	0.16	15.33	11.79	0.85

The decision is much harder here – can I do anything else? One option is to check for common failure cases before passing the string into `Double.Parse`. This means measuring or guessing what the 'common failures' are – an example of such a guess would be to check if the first digit is alphabetic.

Moving on, the C++ case is easier – I can simply return failure from `strtod` by using a return code rather than throwing an exception.

```
bool try_convert(std::string const & arg) {
  const char *p = arg.c_str();
  char *pend = 0;
  (void)strtod(p, &pend);
  return (*pend == '\0');
}
```

Arg	Funct'n #1	Funct'n #2	Funct'n #3	
X	0.17	11.69	0.31	(MSVC unoptimised)
X	0.08	11.03	0.26	(MSVC optimised)

Anything Else?

There are a couple of other points worth noting about using exceptions.

It can be hard to correctly identify which exceptions should be caught, and mismatches can cause other problems.

Firstly, catching too much. If your code catches too broad a category of exceptions (for example "`catch (Exception)`", or "`catch (...)`" in C++) can mean that error cases other than the one you are expecting are caught and do not flow to the appropriate higher level handler where they can be correctly dealt with. This can be even more of an issue in some C++ environments, such as MSVC, where non-C++ exceptions are also swallowed by `catch (...)`.

Conversely, failing to make the exception net wide enough can lead to exceptions leaking out of the function and causing a failure higher up. This has happened to me when using JDBC in Java where the exception types thrown for data conversion errors, such as invalid date format, seem to vary depending on the driver being used.

Debugging exceptions can be a problem. Many debuggers cannot easily filter exceptions, so if your program throws many exceptions it can make the debugging process slow or unwieldy, or swamp output with spurious warnings.

In some environments you can stop when an exception is about to be thrown, but it is very hard to follow the flow of control after that point. The standard flow-of-control mechanisms are usually easier to trace.

Finally the code you write must be exception safe – when exceptions occur you must make sure that the unwinding of the stack up to the catch handler doesn't leave work undone. The main dangers to avoid are leaving objects in inconsistent states and neglecting to release resources. This includes, but is not restricted to, dynamically allocated memory – don't fall for the popular misconception that exception safety is only an issue for C++ programs (see, for example, [1]).

When Are Exceptions Exceptional?

Let's go back to first principles – what are exceptions for?

The exception mechanism can be seen as a way to provide separation of concerns for error handling. It is particularly useful when the code detecting the error is distant from the code handling the error; exceptions provide a structured way of passing information about the error up the call stack to the error handler.

Exceptions also make errors non-ignorable by default, since uncaught exceptions terminate the process. More traditional alternatives such as error return values are often ignored and also the flow of error information has to be explicitly coded which leads to increased code complexity.

Exceptions can, in principle, be viewed as no more than a mechanism to transfer control within a program. However, unless care is taken, using exceptions as a flow of control mechanism can produce obscure code. Stroustrup wrote: *'Whenever reasonable, one should stick to the "exception handling is error handling" view'* [2].

If exceptions are being used for handling errors that need non-local processing then the possible runtime overhead is unlikely to be an issue. Typical uses of exceptions of this type, where errors are relatively uncommon and the performance impact is secondary, include:

- signalling errors which require aborting an entire unit of work, for example an unexpected network disconnection
- support for pre/post conditions and asserts

Grey areas where, since exceptions are thrown for 'non-exceptional' or 'non-error' conditions, programmers disagree about the validity of using exceptions include:

- dealing with invalid user input
- handling uncommon errors in a recursive algorithm – for example a parse failure for a SQL statement or numeric overflow in a calculation
- handling end of file (or, more generally, handling any kind of 'end of data' condition)

Examples of abuse include:

- using exceptions to handle optional fields missing
- using exceptions to give early return for common special cases

My own rough guideline for the 'grey areas' is that if all exceptions became fatal then most programs should still run at least four days out of five.

Others have a more flexible approach and use exceptions more widely than this, sometimes unaware of the consequences of this decision.

Conclusion

It is important to recognise that using exceptions has an associated cost in C# and to a slightly lesser extent in Java and C++.

Using exceptions in the main execution path through the program may have major performance implications. Their use in time-critical software, in particular to deal with non-exceptional cases, should be carefully justified and the impact on performance measured.

When this is an issue alternative techniques which may be faster include: using return values instead of exceptions to indicate 'expected' error conditions; and checking for common failures before risking the exception.

Roger Orr
`rogero@howzatt.demon.co.uk`

Acknowledgements

Thanks to Phil Bass and Alan Griffiths for reviewing earlier drafts of this article.

References

[1] Alan Griffiths, "More Exceptional Java", *Overload*, June 2002

[2] Bjarne Stroustrup, *C++ programming language, 3rd edition*, p375

Where Egos Dare

by Allan Kelly

Recently I have had reason to look through some old software engineering textbooks, the kind of thing I used to read as an undergraduate and junior programmer. This reminded me of a few concepts I haven't thought about in several years. One of these is egoless programming. As I recall my university lecturers were very keen on this concept, it seemed to be *a good thing*. Although, to be honest, I've always had my doubts...

What Exactly is Egoless Programming?

I guess it all depends on what you mean by ego, so I turned to my favourite text on management and psychology (and all that kind of stuff) where I found the following definition:

> *"**ego** has to make sense of the internal conflict in our mind between the id and superego and the external world. The ego is the decision-making part of personality and is engaged in rational and logical thinking. It is governed by the reality principle." (Mullins, 2002)*

Now this looks a bit confusing. Does egoless programming mean programming without rational and logical thinking? Well, I've seen my fair share of programs and I must say a lot of them do seem to lack rational and logical thinking, but I really don't think anyone wants to advocate this as a design and programming technique. What would an irrational program look like? What would an illogical program look like?

I'll admit, the first half of the sentence "conflict in our minds" makes sense, I often experience conflict when I'm programming: do I use `++i` or `i++`, or even `i:=i+1`? And I'm often torn between doing something the "modern" way (say with a template function specialisation) or the "old fashioned way" (with lots of verbose code). So maybe conflict-less programming would be a good idea.

But then, if there was no conflict why do we need people? It is the human judgement, honed over years of programming that makes developers so valuable. If there is no conflict, if there is an obvious solution every time, we can automate it, bring on the Case tools. Or even just enhance the compiler. The fact is, resolving conflicts and balancing competing forces is a fundamental part of what we as software developers do.

So clearly this is not the right definition for ego.

Maybe what the books mean is superego-less programming, if egoless is good, then surely, super-egoless must be better?

> *"**Super ego** is the conscience of the self, the part of our personality which is influenced by significant others in our life. " (Mullins, 2002)*

Well programming without consciousness, that doesn't sound good. Once or twice I've programmed into the wee small hours of the morning and come pretty close to coding in my sleep but I wouldn't recommend it.

I'm sure I've met managers who would prefer it if our significant others didn't influence our lives. A manager once asked me to cancel a holiday, however the thought of how my significant other would react prevented me from agreeing.

So, I don't think superego-less programming is a good thing either. That leaves us with id-less programming.

> *"**id** consists of the instinctive, hedonistic part self.." (Mullins, 2002)*

This sounds more like it. Id-less programming – programming without fun. This is work, this is serious, fun has no place in code. (Erh, why did I get into this business?)

While I've known many managers who don't see fun as an essential part of the job, I can't say any have really objected to a bit of fun. After, a bit of fun, a few smiles in the office, makes the day much more, erh, fun.

Try Again

So I'm not a lot closer to understanding what egoless programming is meant to give me. Maybe I'm reading too much of the definition, maybe what we want is a simpler definition of ego. So this time I turned my dictionary, this definition looks more hopeful:

> *"ego n. pl. egos 1. The part of a person's self that is able to recognise that person as being distinct from other people in things. 2. A person's opinion of his or her own worth: men with fragile egos." (Collins, 2001)*

So maybe egoless means we can't tell ourselves from other people, we lose ourselves in some kind of great group. Let's forget that 20th century literature and popular philosophy emphasise the individual, we want to hire programmers who can't tell themselves from anybody else. That might get really confusing at times, and where is the difference of opinions that can lead to so many useful insights?

Or maybe you want people who think they are worthless, we want programmers who don't have very high opinion of themselves. I can see this be a great position for manager to be in when it comes to the annual pay reviews, or for contract renewal.

Manager: Well then Bob, I'd like to renew your contract for another 3 months

Bob: No, no, you don't want me, remember that bug in my code? John had to fix it last week?

Manager: Yes, I see, well, I can't throw you out on the streets, so what say I keep you for another three months with a 25% reduction?

Bob: I am not worthy

It seems to me that ego is an essential part of people, and even of software people. On the whole it's more fun to work with people who are confident – I'm sure most people would say no if invited to join a team of people racked by self-doubt. Actually, we want programmers with egos. We want programmers who care, we want people who say "I'm proud to have worked on this project."

It seems "egoless programming" doesn't really stand up to analysis. We want people who are rational, logical, proud of their work and bring a positive attitude to work.

Origins and Setting

The term egoless programming originated with Gerald M. Weinberg's "The Psychology of Programming" in 1971. In the Silver Anniversary edition (Weinberg, 1998) and in IEEE Software (Weinberg, 1999) Weinberg has reprised his original ideas and claimed he has been misunderstood and misinterpreted. To be fair, Weinberg was making an argument for teams, it just happened people remembered the sound bite, *egoless programming*, and forgot a lot of his other ideas.

For Weinberg egoless programming is about code reviews and letting others comment on your work. Although the benefits of code reviews are well known they are not always conducted routinely. There are a variety of problems, not least because it often takes longer to review code than it takes to write it in the first place.

In Weinberg's original essay he suggested programmers' egos lead them to hide their code, and protect it, they don't want other people passing comment on it. Does this really happen? There is no shortage of programmers posting their code on SourceForge for all to view.

I think the problem is more the social setting of the review. In reviewing your code, and giving feedback, there is a great capacity to hurt someone's feelings. Receiving feedback can be hard, and it can hurt. Simply being told "think of egoless programming" is like being told to keep "the British stiff upper lip."

Giving criticism so it doesn't hurt, and receiving criticism without feeling personally attacked, are skills themselves. One organisation I know did code reviews by e-mail, a day or two after your check-in you would receive an e-mail from your reviewer listing your mistakes. This may be efficient but it is also brutal. Some developers would actually hold off check-ins until the last minute then make a lot in a short space of time, this usually meant the reviewers could not get the reviews done before the release negating the whole point of a review. I think this kind of humanless code reviewing from 30,000 feet is a bit like carpet bombing, painless for the reviewers but indiscriminate and uncaring.

Neglected Teams

Programming teams are hardly new concepts, they have been knocking around software development books for many years. Ever since programming projects got beyond the abilities of one person we have had teams.

The problem is that our traditional text books devote hundreds of pages to technical issues and almost nothing to team work. A statement to the effect that "much software is developed in teams, good teams need egoless programming" is about as far as many go.

For example, to take a standard textbook; if Pressman (1994) even mentions teams in the text it doesn't make it into the index. His fourth edition (Pressman, 1997) edition is slightly better but with over 800 pages you could easily miss the few pages on teamwork. Other texts can be better, for example Somerville (2001), devotes a whole chapter (20 pages) to managing people, and six pages alone to team working, pretty good going until you notice it is an 800 page book!

So, while we can all agree that teamwork is important few of us actually devote time to thinking about how to make it work. There is an assumption somewhere that teams just work, once told we are a team then all is sweetness and light.

Personally I don't see this myself. Teamworking is a skill just as much as C++ or SQL is, and we need to learn it. In fact, each team needs to relearn the skill itself. This may be especially true of programmers, I know quite a few like myself who preferred the warm dry computer room at school to the cold, wet, muddy football pitch where our teamwork skills where supposed to be developed.

The point is teamwork doesn't just happen, we need to be encouraged to work in teams. It can be scary talking to new people, it can be terrifying trusting somebody else to do work, and it can be demoralising to see somebody else do a piece of work that you really want to do – and I haven't even mentioned fixing people's bugs!

Simply extolling the virtues of teamwork, asking people to practice "egoless programming" doesn't make it happen. If a company wants these objectives it has to work for them.

The Irony

Perhaps the greatest irony of all is that people are social animals, we actually like interacting with other people, working and playing with other people. In fact work can be so much more enjoyable when you work with a great team, people we trust, people we value. When we trust people work becomes so much easier, we don't need to keep an eye on them, we don't need to secretly double check their code, and we feel happy for them to the see our code.

There are times when competing is the right solution. Put two teams on a football field, two companies in the same market, any company and Microsoft! Competition is the bedrock of capitalism, competition drives us, we all want to win. But competition is not always the right answer, sometimes we get more by co-operating then competition.

This is why companies exist, because as a group of co-operating individuals we can achieve something that the same individuals can't achieve by competition. And that is why programming teams exist, because some programs are too big for one person to write.

It Isn't Easy

One of the most difficult things for a team to do is to overcome Conway's Law (Kelly, 2003, Conway, 1968, Coplien, 2003). This is usually stated as:

> *"if n developers work on a compiler, it will be an n pass compiler"*

It is so easy on any project to count the number of developers and divide the project into that number of pieces, even if the project can divided into more, or less pieces. This way we can keep everyone busy, everyone can have their own space and get on with a piece of work.

But is this always sensible? If a project naturally has a front and a back-end why divide it into three pieces just because we have three programmers? The bigger question is: What are we trying to optimise here? Are we just trying to make three programmers look busy?

A common variation on Conway's Law states:

> *"if n developers work on a compiler, it will be an n-1 pass compiler, somebody has to manage "*

This assumes that the role of managers is as police. They are there to command and control the workers (programmers) and ensure they get the work done.

Actually, I think we need to extend Conway's law, it's really Conway's Trap:

> *"The sub-optimal team structure and system design that occurs when n programmers divide any given piece of work into n pieces and allocate one piece per programmer."*

This formula is easy to apply, has a superficial attractiveness and is easy to manage. It's particularly easy to manage if some of the n programmers don't particularly like working in a team or communicating with other programmers. Unfortunately, there is no real reason for dividing the project into n pieces, no reason to believe the n pieces are of equal size, or equal complexity and for the project to complete each n piece must be delivered.

Teamwork

If we want our teams to really work well and develop good software we need to move away from simple exhortations to "egoless programming." We need to break our projects into the optimal development pieces and work as teams, rather than slavishly divide by the number of programmers and work as a group of individuals.

This means we have to think seriously about making our teams work well together. Fortunately some of the more recent writings on software development have started to put a greater emphasis on teamwork (e.g. Eckstein, 2003, Cockburn, 2002) and the (in)famous pair programming (Beck, 2000, Coplien, 2003) is a good example of this. But learning to work as a team doesn't start and stop with pair programming.

It helps if a team actually knows one another. Do they lunch together? Do they socialise together? Some companies try to get teams to socialise by arranging a Friday afternoon "beer bash", these can have an artificial feel to them (especially if everyone is driving home) but is a start.

Other high profile "teambuilding activities" like white-water rafting or paintballing can also be the subject of jokes and mockery. It's important to match your team building efforts to the attitude of the team, maybe a trip to the pub or the cinema is more in keeping with your team. Even ad hoc communal gathering areas such as kitchens can be far more effective than one off events.

This requires ongoing expense and commitment from the company, after all that kitchen space is a continuing cost while a paintball day is a one-off expense. But this long-term commitment is what is required to build a good team, it doesn't happen overnight.

Companies need to look at their own mechanisms: do they reward people for teamwork? Or do they award annual bonuses based on individual heroic coding efforts? And are teams broken up once a project finishes, or can they move together onto the next project? Are people allowed to sit together? Or are people squeezed into whatever space can be found when they arrive on day one?

There is a lot individual managers can do here too. If they see their role as commanding and controlling the developers they aren't going to get the best from them. They need to learn to develop their teams, encourage people to work together and learn together.

Keep Your Ego, Make Teams Work

The fact is we want developers to have egos, we want them to be proud of their work, we want them to think logically and rationally. But we want to harness these egos within a team. We want the team to succeed. This can only happen if we are socially aware and build towards this goal.

We can't expect any of this to happen just because we say it should happen. We need to work hard to make these teams work, people need to learn how to work in teams, how to work with their colleagues, how to give constructive feedback and how to accept feedback.

Unfortunately, neither Microsoft nor Rational sells a tool to do this, its something you have to create yourselves. You can bring in outside help but this is a long process, the rewards are great but it won't happen overnight.

Allan Kelly
`allan@allankelly.net`

Bibliography

Beck, K. (2000) *Extreme Programming Explained*, Addison-Wesley.

Cockburn, A. (2002) *Agile Software Development*, Addison-Wesley.

Collins (2001) *Collins Paperback English Dictionary*, Harper Collins, Glasgow.

Conway, M. E. (1968) *How do committees invent?*, Datamation.

Coplien, J., and Harrison, N. (2003) *Organizational Process Patterns* (forthcoming), `http://www.easycomp.org/cgi-bin/OrgPatterns`, Wiki web site for book

Eckstein, J. (2003) *Scaling Agile Processes* (forthcoming), Dorset House, New York.

Kelly, A. (2003) "The original Conways Law", *Overload*.

Mullins, L. J. (2002) *Management and organisational behaviour*, Prentice Hall.

Pressman, R. S. (1994) *Software Engineering: A practictioner's approach (European Adaptation)*, McGraw-Hill Book Company.

Pressman, R. S. (1997) *Software Engineering: a practioner's approach (European adaptation)*, McGraw-Hill.

Somerville, I. (2001) *Software Engineering*, Pearson Education, Harlow.

Weinberg, G. M. (1998) *The Psychology of Computer Programming*, Dorset House Publishing.

Weinberg, G. M. (1999) *Egoless Programming*, IEEE Software.

The Vision Thing

One way to get a team working together is through shared vision. While "vision" may seem a bit abstract, vague or ephemeral, it does actually have strong supporters who argue that creating a shared vision is a powerful tool for managers and teams:

> *"Where there is a genuine vision (as opposed to the all-too-familiar 'vision statement'), people excel, and learn, not because they are told to, but because they want to." (Senge, 1990)*

There is a brilliant example of the power of vision in an IT project by Conklin. He described the management of the Digital Alpha AXP project in the early 1990s. This project employed over 2,000 engineers both in hardware (chip design, machine design, integration) and software (at least two operating systems, compilers, editors, etc., etc.). He called this *Enrolment Management* and at the centre of it was vision.

This shared vision was not a weak, ephemeral thing but a strong substantial, lasting vision which moved people to produce seemingly extraordinary work:

> *"given the group's commitment to the larger result, we found more aggressive behaviour. For example, the OpenVMS AXP group publicly committed to their target schedule and stated, 'We don't know how to achieve this, but we commit to finding a way'." (Conklin, 1996)*

Enrolment management used a simple four-point methodology:

1. Establish an appropriately large shared vision;
2. Delegate completely and elicit specific commitments;
3. Inspect rigorously, providing supportive feedback;
4. Acknowledge every advance, learning as the program progresses.

The case study describes how the project management admitted they had no project plan. Nor could they possibly draw one up in the time available. Instead they took the difficulties as challenges and used each new problem as an opportunity to enforce the vision and increase the speed of development.

Perhaps most interesting about this is the similarities between many of Conklin's ideas and those of the Agile Development proponents. Perhaps the biggest difference is that most Agile advocates duck the issue of large teams engaging in Agile Development, but Conklin, in 1990, was doing Agile Development with 2,000 engineers.

Conklin, P. F. (1996) "Enrolment Management: Managing the Alpha AXP Program", *IEEE Software*, 13, 53-64.

Senge, P. (1990) *The Fifth Discipline*, Random House Books.

A Template Programmer's Struggles Resolved

by Stefan Heinzmann and Phil Bass

This article is the result of the conversations between the two authors (Phil Bass and Stefan Heinzmann) that were triggered by the latter's article in this very issue of Overload [1]. Stefan originally wanted to have the resolution of the problems outlined in his article published in the next issue in order to keep you in creative suspension for a while, but the Editor found that to be too cruel, so we tried to finish this article in record time.

Article [1] ended with 4 unsolved problems which are repeated here:

- It requires the ugly cast for passing the null pointer as the third argument to `lookup`
- `lookup` returns the result by value, which can be inefficient
- It is still unclear why I couldn't use the `typedef`s from `std::binary_function` in the `LessKey` predicate (only with GCC)
- Neither do I know why the compiler wanted to convert the wrong way between `Val` and `EVal`

The first two problems relate to the `lookup` function template, while the last two problems relate to the `LessKey` predicate. In addition, the text itself lists as a fifth problem: How to ensure at compile time that the lookup table is sorted. Just for a change, let's tackle those problems in reverse order.

Ensuring the Lookup Table is Sorted

Thaddaeus Frogley suggested that rather than sorting the table at compile time, which may be impossible, the debug build could check the sorting when the program starts up. You would need to write a function that checks the table sorting and call it in an assert macro that is run at program startup. As an additional service, the checking code could generate a sorted version of the table in a format that can be cut and pasted into the source code, so that the burden of keeping the table sorted is not on the struggling programmer. We'll not actually show any code for this here, as we believe that it is fairly straightforward. Furthermore, this issue was not the main point in [1] anyway.

The `LessKey` Predicate

Here, we owe you an explanation why the `typedef`s in the `binary_function` base class could not be used in the definition of `LessKey::operator()`. It turns out that this is because of the name lookup rules, namely *two-phase lookup*. If you want to know the full story you need to turn to *The Book* [2] chapter 9.4.2, but here's the bottom line:

As `std::binary_function` is a base class for `LessKey` that depends on `LessKey`'s template parameters, it is called a *dependent base class*. The C++ standard says that nondependent names are *not* looked up in dependent base classes. Hence a standard conforming compiler will not find `result_type` and its siblings and emit an error message. The error messages that were actually emitted by GCC weren't particularly helpful, in particular the mentioning of `typename` was misleading, but the code was definitely wrong. So what can we do about it? There are a number of possibilities here, though none are particularly appealing:

- Dodge the issue in the way Stefan did it, i.e. by not using `result_type` etc. inside `LessKey`. The downside of this is that the fairly complicated element type must be written out several times.
- Explicitly qualify the `typedef` names with the name of the base class. This makes the names dependent, therefore they are looked up and found in the base class. However that removes the whole point of using the `typedef`s, because the base class name is a complicated template.
- Bring the `typedef` names into the scope of the derived class with a using declaration. That doesn't save any typing either, because the base class name needs to be spelled out, too.
- Add another `typedef` to the derived class. That is also fairly verbose, so it may not be an improvement over the first solution.

That's disappointing, isn't it? It means that deriving from `std::binary_function` in order to make the predicate adaptable according to the rules of the STL is only half as useful as you'd wish it to be. It makes you wonder whether you want to derive from `std::binary_function` at all. After all, you can make your predicate adaptable by providing the required `typedef`s yourself, like this:

```
template<typename Key, typename Val>
struct LessKey {
  typedef bool result_type;
  typedef const Pair<Key,Val> &first_argument_type;
  typedef first_argument_type second_argument_type;
  result_type operator()(first_argument_type a,
                         second_argument_type b) const
    { return a.key < b.key; }
};
```

If you don't want the predicate to be adaptable, you can apply a clever trick that appeared in [3]: You make the predicate work on the key type directly. This removes the need to construct an element with all its associated problems mentioned in [1]. If we needn't construct an element, we need no default constructor for the `Val` type either, which is an additional advantage. Here's the code:

```
template<typename Key, typename Val>
struct CleverLessKey {
  typedef const Pair<Key,Val> Elem;
  bool operator()(Elem &elem, const Key &key) const
    { return elem.key < key; }
  bool operator()(const Key &key, Elem &elem) const
    { return key < elem.key; }
};
```

Note the overloading of `operator()` to allow passing the arguments in any order. Inside the lookup function template the key can now be passed directly to `equal_range`:

```
template<typename EKey, typename EVal,
         unsigned n, typename Key, typename Val>
EVal lookup(const Pair<EKey,EVal>(&tbl)[n],
            const Key &key, const Val &def) {
  typedef CleverLessKey<EKey,EVal> Pred;
  typedef const Pair<EKey,EVal> Elem;
  std::pair<Elem*,Elem*> range
      = std::equal_range(tbl, tbl+n, key, Pred());
  if(range.first != range.second)
    return range.first->val;
  return def;
}
```

This bends the intuitive rule of what a good predicate is; ordinarily you would think both argument types had to be the same, but as far as we know there's nothing in the C++ standard that would make this illegal. It certainly works with VC++ 7.1 and GCC 3.3.

The Comeau compiler [4] disagrees, however. Apparently, the library implementation used by Comeau contains compile-time concept checks that verify whether the two argument types of the predicate are the same. As ours are not, those checks fail and the compiler rejects the code. We feel that this is overly restrictive.

That leads us to the weird error mentioned in [1] where the compiler apparently wanted to convert an `int` to a `const char [4]`. We owe you an explanation here, too. Let us repeat the relevant code where the error occurs:

```
template<typename EKey, typename EVal,
         unsigned n, typename Key, typename Val>
EVal lookup(const Pair<EKey,EVal>(&tbl)[n],
            const Key &key, const Val &def) {
  typedef LessKey<EKey,EVal> Pred;
  typedef const Pair<EKey,EVal> Elem;
  Elem entry = { key, Val() };        // error here
  std::pair<Elem*,Elem*> range
     = std::equal_range(tbl, tbl+n, entry, Pred());
  if(range.first != range.second)
    return range.first->val;
  return def;
}
```

The error message was strange, but there's indeed an error. The compiler correctly deduces the following type for `Val`, namely `const char [4]`. That means that `Val()` tries to default construct a temporary of type `const char [4]`, which is impossible. No conversion from `int` to `const char [4]` is involved, the error displayed by the compiler is again rather misleading here.

The fix employed in [1] was correct, but there is another, more elegant possibility:

```
template<typename EKey, typename EVal,
         unsigned n, typename Key, typename Val>
EVal lookup(const Pair<EKey,EVal>(&tbl)[n],
            const Key &key, const Val &def) {
  typedef LessKey<EKey,EVal> Pred;
  typedef const Pair<EKey,EVal> Elem;
  Elem entry = { key };    // second part of Pair
                           // omitted
  std::pair<Elem*,Elem*> range
     = std::equal_range(tbl, tbl+n, entry, Pred());
  if(range.first != range.second)
    return range.first->val;
  return def;
}
```

As you may have noted, it is not actually necessary to explicitly initialize the `val` member of the variable `entry`, as we don't use it anyway. Omitting the initializer for `val` causes it to be value-initialized, which in practice does the same as calling `EVal()` explicitly.

Anyway, we will use our clever predicate henceforth, despite the problems with the Comeau compiler.

The `lookup` Function Template

Now that we've got those niggles out of the way, we can turn to the remaining problems. Remember what the question was: Given the following program fragment, how can we implement the `lookup()` function as a function template that works for arbitrary instantiations of `Pair<Key,Val>`?

```
template<typename Key, typename Val>
struct Pair {
  Key key;
  Val val;
};

const Pair<int, const char*> table[] = {
  { 0, "Ok" },
  { 6, "Minor glitch in self-destruction module" },
  { 13, "Error logging printer out of paper" },
  { 101, "Emergency cooling system inoperable" },
  { 2349, "Dangerous substances released" },
  { 32767, "Game over, you lost" }
};

int main() {
  const char *result = lookup(table,6,(char*)0);
  std::cout << (result ? result : "not found")
            << std::endl;
}
```

The implementation we arrived at in the last chapter still has those two problems:

- It requires an ugly cast for passing the null pointer as the third argument to lookup.
- `lookup` returns the result by value, which can be inefficient.

Replacing `equal_range` by `lower_bound`

First, note that the `lookup()` function given above is based on `std::equal_range()`, but actually behaves more like `std::lower_bound()`. It returns the mapped value of the first element matching the key, if any. Since `lower_bound()` is a simpler and slightly more efficient algorithm we will use it in the subsequent code samples. However, it is a bit trickier to use, as you'll see shortly.

While `equal_range` returns a pair of iterators, `lower_bound` returns just one. If there are suitable elements (according to the predicate) in the collection, the returned iterator points to the first one found. Otherwise it points to the place where such an element could be inserted without breaking the sorting order. This makes it more complicated to test for success. Here's what `lookup` would look like using `lower_bound` instead of `equal_range`:

```
template<typename EKey, typename EVal,
         unsigned n, typename Key, typename Val>
EVal lookup(const Pair<EKey,EVal>(&tbl)[n],
            const Key &key, const Val &def) {
  CleverLessKey<EKey,EVal> pred;
  const Pair<EKey,EVal> *pos
        = std::lower_bound(tbl, tbl+n, key, pred);
  if(pos == tbl+n || pred(key,*pos))
    return def;
  return pos->val;
}
```

Generalization for Arbitrary Maps

Lets take a step back and look at the problem from a generic angle: Why restrict ourselves to arrays? Surely, it should be possible to write `lookup()` so that it works with any map-like container. In particular, we would expect to be able to write:

```
// ... table definition, etc. as before ...

using namespace std;

int main() {
  map<int, const char*> message_map;
  message_map[6]
       = "Minor glitch in self-destruction module";
  cout << lookup(message_map, 6, "not found”)
       << endl;
  cout << lookup(message_map, 6, 0) << endl;
  cout << lookup(table, 6, "not found") << endl;
  cout << lookup(table, 6, 0) << endl;
}
```

Although this is a slightly different problem, it points the way towards a solution to the original problem that removes the remaining blemishes in Stefan's code.

The `lookup` function needs to be implemented in quite different ways for maps and arrays. For a map, `lookup` should call `std::map<>::find()`; for an array, it should call `std::lower_bound()`. These two functions have different interfaces. The former is a member function taking a single parameter; the latter is a non-member function taking four parameters (for the overload we need). Somehow, the `lookup` function template must deduce these differences from the type of the container passed to it.

In principle, we could just provide separate overloads for maps and arrays:

```
template<typename Key, typename Val>
const Val& lookup(const std::map<Key,Val>&,
                  const Key&, const Val&);

template<typename Key, typename Val, unsigned n>
const Val& lookup(const Pair<Key,Val>(&)[n],
                  const Key&, const Val&);
```

But that gets us back to where we came in. Stefan's article was all about the difficulties of implementing the second of those `lookup()` function overloads.

An alternative approach treats lookup as an algorithm that can be applied to any map-like container. There is a single `lookup()` function, but there can be several types of “map”. Or, more precisely, we define a generic Map concept, write the `lookup()` function template in terms of the Map interface and provide as many implementations of the Map concept as we need (including `std::map<>`s and arrays of `Pair<>`s).

Using the Map concept, the lookup algorithm would look something like this:

```
// pseudo-code
template<typename Map>
const MapVal& lookup(const Map&, const MapKey&,
                     const MapVal&);
```

Here, we have used `MapKey` and `MapVal` to stand for the Map's key and mapped value types. In real C++ code these types would be deduced from the Map type. In the case of `std::map<>` the `MapKey` and `MapVal` types are immediately available: `MapKey` is `std::map<>::key_type` and `MapVal` is `std::map<>::mapped_type`. In the case of arrays things are less straightforward. Arrays are not classes, so we can't add nested `typedef`s. Instead we can use the *traits class technique*.

```
template<typename Map>
const typename map_traits<Map>::mapped_type&
lookup(const Map& map,
       const typename map_traits<Map>::key_type&
                   target_key,
       const typename map_traits<Map>::mapped_type&
                   default_value);
```

Now we can put information about the differences between maps and arrays in the traits class and use that information in our implementation of `lookup()`.

Note that now there's only one template parameter for the compiler to deduce: The type of the map itself. Only the first argument to the `lookup` function participates in this deduction, as the types of the other arguments are dependent on the result of this deduction. This greatly reduces the chance for deduction problems such as ambiguities.

Implementation of `lookup()` Traits

As noted above, `lookup()` will need to call either `std::map<>::find()` or `std::lower_bound()`. A `map_traits<>::find()` function is introduced to hide this from the `lookup()` function itself. Then `lookup()` needs to check whether the key was found and return either the default value or the value part of the element matching the key. Here, we use a `map_traits<>::end()` function to get the appropriate past-the-end iterator for the ‘key found’ test. Retrieving the mapped value via an iterator depends on the element type (not the map type), so an element traits template with a `mapped_value()` member is used for that.

The traits functions represent operations that might be useful in other contexts. In those cases writing

```
map_element_traits<Map>::mapped_value(element)
```

for example, is both tedious and verbose. So, we provide non-member wrapper functions to simplify the code in these situations and take advantage of them in the `lookup()` function itself.

```
// The mapped_value() convenience function
template<typename Elem>
inline
const typename map_element_traits<Elem>::mapped_type&
mapped_value(const Elem& element) {
  return map_element_traits<Elem>::mapped_value(
                                          element);
}

// The find() convenience function
template<typename Map>
inline
typename map_traits<Map>::const_iterator
find(const Map& map,
     const typename map_traits<Map>::key_type& key) {
  return map_traits<Map>::find(map, key);
}
```

```
// The lookup() function
template<typename Map>
const typename map_traits<Map>::mapped_type&
lookup(const Map& map,
       const typename map_traits<Map>::key_type&
                   target_key,
       const typename map_traits<Map>::mapped_type&
                   default_value) {
  typename map_traits<Map>::const_iterator i
        = find(map, target_key);

  return (i == end(map))
          ? default_value : mapped_value(*i);
}
```

The traits templates are declared (but not defined) and then specializations are defined for each of the map-like containers we wish to support. This prevents the instantiation of the traits templates with unexpected template arguments (e.g. via template argument deduction), which should help to minimize the number of incomprehensible error messages if the programmer makes a mistake.

```
template<typename Elem> struct map_element_traits;
template<typename Map> struct map_traits;
```

Traits for `std::map<>`

The traits templates for `std::map<>` are straightforward. A partial specialisation of the map element traits template is provided for any `std::pair<>`.

```
template<typename Key, typename Val>
struct map_element_traits< std::pair<Key,Val> > {
  typedef std::pair<Key,Val> value_type;

  typedef typename value_type:: first_type key_type;
  typedef typename value_type::second_type
                                    mapped_type;

  static const key_type& key(
                  const value_type& element) {
    return element.first;
  }

  static const mapped_type& mapped_value(
                  const value_type& element) {
    return element.second;
  }
};
```

Similarly, a partial specialization of the map traits template is provided for any `std::map<>`.

```
template<typename Key, typename T, typename Cmp,
         typename A>
struct map_traits< std::map<Key,T,Cmp,A> > {
  typedef std::map<Key,T,Cmp,A> map_type;

  typedef typename map_type::key_type key_type;
  typedef typename map_type::mapped_type mapped_type;
  typedef typename map_type::value_type value_type;
  typedef typename map_type::const_iterator
                                  const_iterator;
  static const_iterator begin(const map_type& map)
    { return map.begin(); }
  static const_iterator end(const map_type& map)
    { return map.end(); }

  static const_iterator find(const map_type& map,
                             const key_type& key)
    { return map.find(key); }
};
```

These traits classes adapt `std::map` for use with our `lookup()` function.

Traits for Arrays of Key,Value Pairs

The traits for arrays of `Pair<Key,Val>` are similar to those for `std::map<>`. There is a partial specialisation of `map_element_traits<>` for `Pair<Key,Val>`.

```
template<typename Key, typename Val>
struct map_element_traits< Pair<Key,Val> > {
  typedef Pair<Key,Val> value_type;
  typedef Key key_type;
  typedef Val mapped_type;

  static const key_type& key(
                      const value_type& element)
    { return element.key; }

  static const mapped_type& mapped_value(
                      const value_type& element)
    { return element.val; }
};
```

And there is a partial specialisation of `map_traits<>` for arrays of `Pair<Key,Val>`.

```
template<typename Key, typename Val, unsigned n>
struct map_traits< Pair<Key,Val>[n] > {
  typedef Key key_type;
  typedef Val mapped_type;
  typedef Pair<Key,Val> value_type;
  typedef const value_type* const_iterator;

  static const_iterator begin(
                  const value_type (&map)[n])
    { return &map[0]; }
  static const_iterator end(
                  const value_type (&map)[n])
    { return &map[n]; }

  static const_iterator find(
                  const value_type (&map)[n],
                  const key_type& target_key) {
    const_iterator i = std::lower_bound(begin(map),
                 end(map), target_key,
                 key_value_compare<value_type>());

    return (i == end(map) || key(*i) != target_key)
            ? end(map) : i;
  }
};
```

The find() function uses lower_bound() passing a key comparison predicate with two asymmetric function call operators. The predicate class is generated from the following template:

```
template<typename Elem>
struct key_value_compare {
  typedef typename
            map_element_traits<Elem>::key_type
            key_type;
  typedef typename
            map_element_traits<Elem>::value_type
            value_type;

  bool operator()(const value_type& x,
                  const value_type& y) const {
    return map_element_traits<Elem>::key(x) <
           map_element_traits<Elem>::key(y);
  }

  bool operator()(const value_type& elem,
                  const key_type& key) const {
    return map_element_traits<Elem>::key(elem) <
           key;
  }

  bool operator()(const key_type& key,
                  const value_type& elem) const {
    return key <
           map_element_traits<Elem>::key(elem);
  }
};
```

This is a generalisation of the CleverLessKey class shown above. It uses the key() function from the map element traits to ensure that the predicate class can be generated for any element of a map-like class and only those elements. A third operator() overload is provided so that two elements can be compared to each other. We omitted the additional typedefs needed for adaptability.

Problem Solved! - Problem Solved?

The code presented in the previous sections does fix all the imperfections in Stefan's version. At least, it does if you are using the GCC compiler (Code tested on gcc 3.2 and 3.3.). But VC++ 7.1 produces an error. It turns out that it fails to find the map_traits specialization for Pair<Key,Val>. The reason is related to const qualification. VC++ is happy if the map_traits template is specialized for const Pair<Key,Val> instead of just Pair<Key,Val>, but that creates an error when compiled with GCC. We don't know yet whether this is because of a compiler error or because of an imprecision in the C++ standard. In practice, you will therefore have to provide both (otherwise identical) specializations.

So here is the entire code in all its glory:

```
#include <iostream>
#include <algorithm>
#include <map>
#include <functional>
```

```
// Generic map element declarations.
template<typename Elem> struct map_element_traits;

template<typename Elem> inline
const typename map_element_traits<Elem>::mapped_type&
mapped_value(const Elem& element) {
  return map_element_traits<Elem>::mapped_value(
                                              element);
}

template<typename Elem> inline
const typename map_element_traits<Elem>::key_type&
key(const Elem& element) {
  return map_element_traits<Elem>::key(element);
}

template<typename Elem>
struct key_value_compare {
  typedef typename
            map_element_traits<Elem>::key_type
            key_type;
  typedef typename
            map_element_traits<Elem>::value_type
            value_type;

  bool operator()(const value_type& x,
                  const value_type& y) const {
    return map_element_traits<Elem>::key(x) <
           map_element_traits<Elem>::key(y);
  }

  bool operator()(const value_type& elem,
                  const key_type& key) const {
    return map_element_traits<Elem>::key(elem) <
           key;
  }

  bool operator()(const key_type& key,
                  const value_type& elem) const {
    return key <
           map_element_traits<Elem>::key(elem);
  }
};

// Generic map declarations.
template<typename Map> struct map_traits;

template<typename Map> inline
typename map_traits<Map>::const_iterator
find(const Map& map,
     const typename map_traits<Map>::key_type&
                     key) {
  return map_traits<Map>::find(map, key);
}

template<typename Map> inline
typename map_traits<Map>::const_iterator
end(Map const& map) {
  return map_traits<Map>::end(map);
}
```

```
// The lookup() function
template<typename Map>
const typename map_traits<Map>::mapped_type&
lookup(const Map& map,
       const typename map_traits<Map>::key_type&
                 target_key,
       const typename map_traits<Map>::mapped_type&
                 default_value) {
  typename map_traits<Map>::const_iterator pos
      = find(map, target_key);

  return (pos == end(map))
          ? default_value
          : mapped_value(*pos);
}

// specializations for std::map
template<typename Key, typename Val>
struct map_element_traits< std::pair<Key,Val> > {
  typedef std::pair<Key,Val> value_type;

  typedef typename value_type:: first_type
              key_type;
  typedef typename value_type::second_type
              mapped_type;

  static const key_type& key(
                      const value_type& element) {
    return element.first;
  }

  static const mapped_type& mapped_value(
                     const value_type& element) {
    return element.second;
  }
};

template<typename Key, typename T, typename Cmp,
         typename A>
struct map_traits< std::map<Key,T,Cmp,A> > {
  typedef std::map<Key,T,Cmp,A> map_type;

  typedef typename map_type::key_type
              key_type;
  typedef typename map_type::mapped_type
              mapped_type;
  typedef typename map_type::value_type
              value_type;
  typedef typename map_type::const_iterator
              const_iterator;

  static const_iterator begin(const map_type& map)
    { return map.begin(); }
  static const_iterator end(const map_type& map)
    { return map.end(); }

  static const_iterator find(const map_type& map,
                             const key_type& key)
    { return map.find(key); }
};
```

```
// Our own Pair type suitable for aggregate
// initialization
template<typename Key, typename Val>
struct Pair {
  Key key;
  Val val;
};

// Specializations for Pair
template<typename Key, typename Val>
struct map_element_traits< Pair<Key,Val> > {
  typedef Pair<Key,Val> value_type;
  typedef Key key_type;
  typedef Val mapped_type;
  static const key_type& key(
                    const value_type& element)
    { return element.key; }
  static const mapped_type& mapped_value(
                    const value_type& element)
    { return element.val; }
};

template<typename Key, typename Val, unsigned n>
struct map_traits< Pair<Key,Val>[n] > {  // for GCC
  typedef Key key_type;
  typedef Val mapped_type;
  typedef Pair<Key,Val> value_type;
  typedef const value_type* const_iterator;

  static const_iterator begin(
                    const value_type (&map)[n])
    { return &map[0]; }
  static const_iterator end(
                    const value_type (&map)[n])
    { return &map[n]; }
  static const_iterator find(
                    const value_type (&map)[n],
                    const key_type& target_key) {
    const_iterator i = std::lower_bound(
                  begin(map), end(map), target_key,
                  key_value_compare<value_type>());
    return (i == end(map) || key(*i) != target_key)
            ? end(map) : i;
  }
};

template<typename Key, typename Val, unsigned n>
struct map_traits< const Pair<Key,Val>[n] > {
// for VC++
  typedef Key key_type;
  typedef Val mapped_type;
  typedef Pair<Key,Val> value_type;
  typedef const value_type* const_iterator;

  static const_iterator begin(
                    const value_type (&map)[n])
    { return &map[0]; }
  static const_iterator end(
                    const value_type (&map)[n])
    { return &map[n]; }
    static const_iterator find(
                      const value_type (&map)[n],
                      const key_type& target_key) {
      const_iterator i
            = std::lower_bound(
                   begin(map), end(map), target_key,
                   key_value_compare<value_type>());

      return (i == end(map) || key(*i) != target_key)
              ? end(map) : i;
    }
  };

  // Test code
  typedef const Pair<int, const char*> Elem;

  Elem table[] = {
    { 0, "Ok" },
    { 6, "Minor glitch in self-destruction module" },
    { 13, "Error logging printer out of paper" },
    { 101, "Emergency cooling system inoperable" },
    { 2349, "Dangerous substances released" },
    { 32767, "Game over, you lost" }
  };

  using namespace std;

  int main() {
    map<int, const char*> message_map;
    message_map[6]
         = "Minor glitch in self-destruction module";
    const char *result
         = lookup(message_map, 6, "not found");
    cout << "lookup(map, 6, \"not found\") = "
         << result << endl;
    result = lookup(message_map, 6, 0);
    cout << "lookup(map, 6, 0) = " << result
         << endl;
    result = lookup(table, 5, "not found");
    cout << "lookup(table, 5, \"not found\") = "
         << result << endl;
    result = lookup(table, 6, 0);
    cout << "lookup(table, 6, 0) = "
         << result
         << endl;
  }
```

Afterwords

Phil's Afterword

First, I completely agree with Stefan that C++ is too big, complicated and difficult to use for most programmers and most organisations. C++ is a great language for developing high quality, flexible and efficient software components - but (and I've said this before) most software is not like that.

The main lesson we can take from this apparently simple exercise is that experience counts. I became involved in Stefan's problem because I felt instinctively that the first step should have been to *reduce* the number of template parameters, not (as Stefan tried to do) to increase it. How did I know this would help? I'd been there before.

My advice to C++ programmers struggling with templates (or any other part of the language) is this: learn the rules, don't guess; don't panic; simplify the problem as far as you can; think carefully; experiment; and, finally, don't be afraid to ask for help.

If there is a moral to this story I would say it is that software is a very young discipline. As a profession we still have a lot to learn. One of the problems that remains unsolved is how to design a programming language that is easy to learn, is easy to use, is applicable to a wide range of applications and which generates compact and efficient code. In my opinion, C++ is about the state of the art. It makes most of this possible, but it's not always easy.

Phil Bass
`phil@stoneymanor.demon.co.uk`

Stefan's Afterword

So we've solved all the problems. And I've learned a lot in the process! That's a happy end, isn't it?

I don't think so. Look at what I wanted to achieve at the beginning and what we ended up with. All this is just a clever way to call `std::lower_bound`, isn't it? (Or the `find` member function of `std::map`). Ok, I'm a bit sarcastic here.

If we subtract the test code and the code related to `std::map`, which don't really count here, we have written in excess of 100 lines of source code, some of it quite tricky. And most of it is just the scaffolding needed to make `std::lower_bound` usable in a nice way with constant ROMable key/value pairs. If we include all compiler quirks and crappy error messages, this matter was suitable for filling a fair number of Overload pages.

Clearly there's something amiss here. If this sort of thing does not get easier a lot of programmers will get frustrated by C++.

Stefan Heinzmann
`stefan_heinzmann@yahoo.com`

Editor's Afterword

Reading Stefan's contributions to this issue brought back memories for me of an article I wrote nine years ago (Overload 8). There isn't space to reproduce it in this issue, or the editorial response it elicited (this was longer than the article). I won't go into the detail of the arguments, but just quote from the end of that response:

"At the end of the day, I basically agree with Alan - C++ is harder to use than C - and I think his comparison between a Stylophone and a violin is well drawn. I don't blame the language (and I don't really think Alan does either) - I blame IT management for giving everyone a violin and saying "right, now play a tune!" What C++ highlights is the need for better training, better tools and more realistic expectations." - *Sean A Corfield*

Nine years have passed and nothing significant has changed. C++ is still to hard to use, lacks decent tools and expectations are seldom realistic.

Alan Griffiths

References

[1] Stefan Heinzmann: "The tale of a struggling template programmer", this issue of *Overload*

[2] D. Vandevoorde, N. M. Josuttis: *C++ Templates: The complete guide*, Addison-Wesley 2003

[3] Rich Sposato: "A More Flexible Container", *Overload* issue 58 December 2003

[4] `http://www.comeaucomputing.com`