contents

credits & contacts

Editor:

John Merrells,
merrells@acm.org
241 Heartwood Lane,
Mountain View,
CA 94041-11836, U.S.A

Advertising:

Pete Goodliffe, Chris Lowe
ads@accu.org

Membership:

David Hodge,
membership@accu.org
31 Egerton Road
Bexhill-on-Sea, East Sussex
TN39 3HJ, UK

Readers:

Ian Bruntlett
IanBruntlett@antiqs.uklinux.net

Phil Bass
phil@stoneymanor.demon.co.uk

Mark Radford
twonine@twonine.demon.co.uk

Thaddaeus Frogley
t.frogley@ntlworld.com

Richard Blundell
richard.blundell@metapraxis.com

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

Membership fees and how to join:

Basic (C Vu only): £15
Full (C Vu and Overload): £25
Corporate: £80
Students: half normal rate
ISDF fee (optional) to support Standards work: £21
There are 6 journals of each type produced every year.
Join on the web at www.accu.org with a debit/credit card, T/Polo shirts available.
Want to use cheque and post - email membership@accu.org for an application form.
Any questions - just email membership@accu.org.

Editorial

by Alan Griffiths

Why am I writing the editorial?

Six years ago I edited a single issue (19) of Overload to bridge the gap when the then editor (Sean Corfield) gave up the job and we were seeking a new editor. On that occasion I edited a single issue on the understanding that I would not be able to continue and in the hope that someone would be able to take over. Fortunately John Merrells volunteered to edit the next three issues and, as you will be aware, he is still here!

In the editorial of "my" issue of Overload I admitted that I'd like to be able to do the job on a long term basis but, at that time, there were too many demands upon my time for me to make that commitment. I even went so far as to say that it would be a possibility "in a few years". But, in the meantime, John has retained his commitment and I found standing for the ACCU Chair placed other demands on my time.

Anyway, having now given up the Chair I now have some time I can commit to Overload and have joined the team as a "Contributing Editor". John and I haven't discovered what that title means in practice yet, but we've decided it means that I can write editorials and John doesn't have to.

Six years on

Looking back to the editorial I wrote then makes me aware of quite how much Overload has changed during John's time as editor. While I think all the changes are for the better, it also raises the question of how it will change in the future.

One of the editorial concerns at that time was the range of material that Overload covered. When John took over, Overload was the journal of the ACCU's "C++ Special Interest Group" and was very much focussed on C++. When John took over he began expanding the range of material Overload published beyond C++ - and we regularly have articles on other languages and on other aspects of software development such as design and development methods. While this was good for Overload it did raise questions as to its relationship to the C++ SIG and, after a while, I (as C++ SIG organiser) severed the connection - allowing Overload to be repositioned as the ACCU journal for full members.

Despite having been dissociated from the C++ SIG the majority of Overload material continues to use C++. However, I feel the focus of such articles has changed: there is a tendency for them to be about designs, illustrated using C++, rather than about C++ itself. That is good, because C++ is an extremely expressive language, which continues to surprise and delight me (although I still have the concerns I expressed in Overload 7 about the demands it makes of developer skills).

The current C Vu editor (James Dennett) will recognise the situation John found himself in when he took on Overload: the previous editor had invested a lot of energy into the journal and had done everything (soliciting articles, reviewing them, and editing the journal) himself. John successfully introduced an innovation: he has built up a team of "readers" who work with the authors before publication by reviewing the articles (and making helpful suggestions). (One of the reasons I prefer writing for Overload to writing for C Vu is the feedback I get from the readers prior to publication.) The readers also help John decide what is suitable for publication. The value of having a team working on the journal proved itself when John had to take a break from editing and Einar Nilsen-Nygaard took over for a few issues with no break in the continuity.

Another change is perhaps the most obvious and also the easiest to overlook: the appearance of the journal. While the value of the journal is still in the material the improvements to the appearance from a professional production process are spectacular.

All of this makes Overload a much more impressive publication than it was six years ago.

The future

Over six years I've found that the work I'm doing has changed and that my interests have changed with it. Six years ago I was using C++ to create bits of software that worked without the author being present. Nowadays, I'm trying to create a software development process that works when I'm not there to keep things progressing. In both cases the biggest problem seems to be people that expect hard problems to have easy answers. I've also found that similar techniques are applicable: like using an informal "pattern language" to explain to managers why the fastest developer on a project might be a liability - and what to do about it. (But is this type of material of interest to Overload readers?)

Six years ago the lack of material led to two issues (17/18) being rolled into a single cover. The recent pleas for contributions indicate that this is still a risk. One thing that remains the same is the voluntary nature of the contributions and editing of the material. Those that do contribute are well aware of the benefits, but there has always been a need for new blood. If you feel like seeing your words in print then please get in touch - the Overload team is ready to help!

Alan Griffiths
alan@octopull.demon.co.uk

A bin Manipulator For IOStreams

by Dietmar Kuehl

The standard stream classes support different bases when doing formatted I/O with integers: there are manipulators `std::oct`, `std::dec`, and `std::hex` for octal, decimal, and hexadecimal I/O, respectively. There is, however, no manipulator for writing and reading integers using other bases, although something like base two seems to be a natural choice, too.

The question is thus what manipulators are and what a manipulator for formatting integers using base two would look like. For this discussion, it is sufficient to concentrate on manipulators without arguments. These are quite simple: A manipulator without argument is (at least normally) just a function with a certain signature. For example, `std::hex` looks something like this:

```
namespace std {
  std::ios_base& hex(std::ios_base& ib) {
    ib.setf(std::ios_base::hex,
            std::ios_base::basefield);
    return ib;
  }
}
```

This function just clears a bunch of bits (namely those which are set in `basefield`) in the formatting flags and then sets a few of them again (namely those which are set in `hex`). The standard's formatting functions for integers interpret these flags to determine how integers are to be formatted and act accordingly. However, these functions only work correctly for the bases decimal, octal, and hexadecimal (well, at least these are the only bases for which they are guaranteed to work).

Before going more into the formatting flags let's discuss how these manipulator functions actually work. The above function is used to manipulate the stream with an expression like this:

```
std::cout << std::hex;
```

What happens is actually pretty simple: the shift operator is overloaded to take arguments with the signature `std::ios_base&(*)(std::ios_base&)` and this overload just calls the corresponding function, i.e. something like this (actually, this function is implemented as a template but this would only obfuscate the issue):

```
std::ostream&
std::ostream::operator<<
   (std::ios_base&(*m)(std::ios_base&)) {
  m(*this);
  return *this;
}
```

That is, if you want to implement a manipulator, you would just implement a function with the appropriate signature: it takes a stream (or one of its base classes: `std::ios_base` or `std::ios`) as argument and returns this argument again (the argument and return type have to be identical but there is some choice toward the argument types you can use). The function would just do the manipulation on the argument and then return the argument.

To implement a manipulator which modifies all integer output to become binary is, however, non-trivial because it requires interfering with how integers are formatted and the standard routines for this are not prepared to support bases other than 8, 10, and 16. However, it is doable because it is possible to supply the formatting code for integers by implementing a class derived from the `std::num_put` facet which is then installed (when necessary) by the manipulator.

Formatting Integers

My guess is that talking about facets is somewhat confusing so let's walk through this whole thing, although most of the stuff will not be related to manipulators directly (some additional stuff on manipulators will, however, come up below).

Facets are a means to adapt certain stuff to local conventions. For example, in Germany we use "," as a decimal point and "." as a thousands separator while in other (weird) places, "." is used as a decimal point and "," as a thousands separator. To adapt output (and other stuff) to the conventions the user is used to, the C++ library uses "facets" which are just classes obeying a few requirements (essentially, each object has to have a public

---

Copy Deadlines

All articles intended for publication in *Overload 56* should be submitted to the editor by July 1st, and for *Overload 57* by September 1st.

member of type `std::locale::id` named `id` and all public functions should be `const`, i.e. the objects should be immutable). There are several of them but the interesting one here is `num_put`: the facet doing numerical formatting. Actually, this is not a class but a class template. I will use a template here because it is less confusing than using the specialization we will need to install later.

To replace the functions for formatting integers, a class is derived from `num_put` and a few functions are overridden:

```
template <typename cT, typename OutIt>
class bin_num_put: public
std::num_put<cT, OutIt> {
  OutIt do_put(OutIt to,
               std::ios_base& fmt,
               cT fill,
               long v) const;

  OutIt do_put(OutIt to,
               std::ios_base& fmt,
               cT fill,
               unsigned long v) const;
};
```

There are just two functions dealing with integer values, one for signed and one for unsigned ones. Each function gets an output iterator as argument to write individual characters to. After writing the characters the iterator is returned as the result of the function. The second parameter is a reference to an object holding formatting information. The third parameter is the character to be used for padding a value to a specific length. The last parameter is the value to be formatted.

Since I'm mostly interested in getting the principle right, I will just stick to a rather simple implementation using a fixed width of digits. A "real" implementation might want to omit leading zeros (this isn't really hard to implement either). Effectively, just one formatting function is needed because formatting of signed values can be delegated to formatting unsigned values in this case. The formatting functions will use another facet, `ctype`, to convert the characters `0` and `1` to values of the appropriate character type:

```
template <typename cT, typename OutIt>
OutIt
bin_num_put<cT, OutIt>::do_put(OutIt to,
                std::ios_base& fmt,
                cT fill,
                long v) const {
  return do_put(to, fmt, fill,
         static_cast<unsigned long>(v));
}

template <typename cT, typename OutIt>
OutIt
bin_num_put<cT, OutIt>::do_put(OutIt to,
                std::ios_base& fmt,
                cT fill,
                unsigned long v) const {
  char narrow[] = "01";
  cT wide[2] = { 0 };
  std::use_facet<std::ctype<cT> >(
      fmt.getloc()).widen(begin(narrow),
                end(narrow) - 1,
                begin(wide));
  cT buffer[std::numeric_limits<
                unsigned long>::digits];
  std::fill(begin(buffer),
            end(buffer),
            wide[0]);
  cT* end = end(buffer);
  for (; v != 0; v /= 2)
    *--end = wide[v % 2];

  return std::copy(begin(buffer),
                   end(buffer), to);
}
```

The above code uses the following two auxiliary functions to get an iterator (in this case actually just a pointer) to the beginning and the end of a statically sized array:

```
namespace {
  template <typename T,
            int sz> T* begin(T (&a)[sz]){
    return a;
  }

  template <typename T,
            int sz> T* end(T (&a)[sz]) {
    return a + sz;
  }
}
```

If you don't understand these two functions, just don't worry about them: I'm using them to conveniently get iterators to the beginning and the end of a statically sized array.

The above code explicitly excludes the last element of the array `narrow` (this is what the `-1` is good for). The reason for this is that the array `narrow` has the size three: the null character at the end of the string is included in the array. That is, the line initializing `narrow` is equivalent to this one:

```
char narrow[] = { '0', '1', 0 };
```

The actual formatting of the binary number is trivial: an array with sufficient zeros is obtained and it is filled with digits starting from the end until there are no further digits. This approach also works for formatting integers for bases other than two and this will be used below. Once all digits are available, the array is copied to its destination. The only somewhat tricky part is getting the appropriate characters representing `0` and `1` because we don't really know the character type. This is done by using the facet `std::ctype<cT>` which can "widen" a narrow character to the corresponding character type.

The `bin` Manipulator

OK, now that the routines for formatting integers as binary are implemented, they need to be installed into our stream to have them used. Before implementing a corresponding manipulator,

lets do this in a small test program. The facets are objects held by a "locale" and it is necessary to construct a locale object with the default `num_put` facet replaced by the new facet. To provide this, it is necessary to instantiate the `bin_num_put` class template with appropriate types, i.e. with `char` as character type and `std::ostreambuf_iterator<char>` as iterator type: these are the types used when doing numeric formatting using `std::cout`. If a wide character stream like `std::wcout` is used, `char` has to be replaced by `wchar_t`, of course. Once the new locale is constructed from an existing locale and the `bin_num_put` class, it is installed into the corresponding stream using the `imbue()` function:

```
int main() {
  typedef std::ostreambuf_iterator<char>
                                iterator;
  std::locale loc =
      std::locale(std::cout.getloc(),
       new bin_num_put<char, iterator>());
  std::cout.imbue(loc);
  std::cout << 10 << "\n";
}
```

We can just put the code installing the binary formatting into a manipulator using an appropriate template to make it feasible with all kinds of streams:

```
template <typename cT, typename traits>
std::basic_ios<cT, traits>&
bin(std::basic_ios<cT, traits>& ios) {
  typedef std::ostreambuf_iterator<cT>
                                iterator;
  std::locale loc =
      std::locale(ios.getloc(),
        new bin_num_put<cT, iterator>());
  ios.imbue(loc);
  return ios;
}
```

... and use it in the expected way:

```
std::cout << bin << 10 << "\n";
```

Well, except that there is no easy way to turn binary formatting off again! Since we have replaced the formatting routine we can use the `std::hex` manipulator as often as we want: there will be no change at all.

```
std::cout << bin << 10 << std::hex
          << 10 << "\n"; // does not work
```

To do something like this, it is necessary to take the value of formatting flags into account and act correspondingly. Before supporting use of the standard manipulators it is, however, useful to adapt the case to cope with arbitrary bases.

Storing Formatting Information

To do this, the selected base should be stored with the stream such that it can be used by the formatting function. The obvious place to store such information is in an `iword()` of the `fmt` member. Here are corresponding manipulators which also install the needed facet only if it is not yet present:

```
static int base_index =
                std::ios_base::xalloc();

template <typename cT, typename traits>
std::basic_ios<cT, traits>&
install_bin(std::basic_ios<cT,
                           traits>& ios,
            int base) {
  ios.iword(base_index) = base;

  typedef std::ostreambuf_iterator<cT>
                                iterator;
  if(!dynamic_cast
      <bin_num_put<cT, iterator> const*>(
        &std::use_facet<std::num_put<cT,
             iterator> >(ios.getloc())))
    ios.imbue(std::locale(ios.getloc(),
       new bin_num_put<cT, iterator>()));
  return ios;
}

template <typename cT, typename traits>
std::basic_ios<cT, traits>&
bin(std::basic_ios<cT, traits>& ios) {
  return install_bin(ios, 2);
}

template <typename cT, typename traits>
std::basic_ios<cT, traits>&
oct(std::basic_ios<cT, traits>& ios) {
  return install_bin(ios, 8);
}

template <typename cT, typename traits>
std::basic_ios<cT, traits>&
dec(std::basic_ios<cT, traits>& ios) {
  return install_bin(ios, 10);
}

template <typename cT, typename traits>
std::basic_ios<cT, traits>&
hex(std::basic_ios<cT, traits>& ios) {
  return install_bin(ios, 16);
}
```

The function `xalloc()` "allocates" a new index for formatting information in the stream objects. This index can be used with the `iword()` function of streams: this function returns a reference to an integer. This integer is associated with the stream. Initially, the value returned is set to zero but the above code does not take advantage of this feature. If an integer is not sufficient, a pointer to the formatting information can be stored using the `pword()` function.

There is a function called by the various manipulators which sets the corresponding base and checks whether the appropriate facet is installed. This is done by obtaining the currently installed facet and testing whether it is an instantiation of `bin_num_put`. If it is not,

this facet is installed. What remains to be done is to use the base in the facet, too. The modified code looks like this:

```
template <typename cT, typename OutIt>
OutIt
bin_num_put<cT, OutIt>::do_put(OutIt to,
                 std::ios_base& fmt,
                 cT fill,
                 unsigned long v) const {
  char narrow[] = "0123456789abcdef";
  cT wide[16] = { 0 };
  std::use_facet<std::ctype<cT> >(
      fmt.getloc()).widen(begin(narrow),
           end(narrow) - 1, begin(wide));

  cT buffer[std::numeric_limits<unsigned
                          long>::digits];
  std::fill(begin(buffer), end(buffer),
                            wide[0]);

  int base = fmt.iword(base_index);
  for (cT* it = end(buffer)
        ; v != 0
        ; v /= base)
    *—it = wide[v % base];

  return std::copy(begin(buffer),
                   end(buffer),
                   to);
}
```

Now the manipulators can be tested. For example:

```
int main(int ac, char* av[]) {
  int val = ac == 1 ? 10
                    : std::atoi(av[1]);
  std::cout << "bin: " << bin << val
            << "\n";
  std::cout << "oct: " << oct << val
            << "\n";
  std::cout << "dec: " << dec << val
            << "\n";
  std::cout << "hex: " << hex << val
            << "\n";
}
```

This code is not yet perfect. Actually, several things need to be handled but these are relatively simple and don't need specific new knowledge of the standard library. In particular, the following aspects are not yet addressed but would need handling in a reasonable implementation:

- Negative values conventionally use a minus sign followed by the absolute value rather than the two's complement. That is, the function taking a `long` as argument cannot directly use the `unsigned long` version, at least not for negative decimal values.
- Although quite usual for binary values, leading zeros are normally stripped for other bases. To get leading zeros for binary values while omitting them for other bases, the `width()` currently installed in the stream could be used.
- The formatting has to take care of padding, i.e. it has to add fill characters: if `width()` is non-zero, there should be at least that many characters written to the sequence. Padding is a little bit tricky because there are three possible places where padding, i.e. copies of the `fill` argument, should go:
  - to the left of the value
  - the right of the value
  - between a leading sign and the value or to the left if there is no sign

This is specified by `fmt.flags() & std::ios_base::adjustfield()`: the corresponding values are `left`, `right`, and `internal`. In any case, after the formatting, the `width()` should be set to `0`.

Arbitrary Bases

Of course, most of the formatting issues could be taken care of by the base class: the `do_put()` function could check whether the base is 2 and if it is not delegate processing to the base class. On the other hand, the above facet is capable of formatting integers according to arbitrary bases as long as the base is bigger than 1 and there are sufficient different characters configured to represent the digits. A manipulator setting an arbitrary base would, however, require a parameter. The approach to manipulators with parameters is to just provide a class with a suitable constructor and a shift operator:

```
struct setbase {
  setbase(int base): mBase(base) {}
  int mBase;
};

template <typename cT, typename traits>
std::basic_ostream<cT, traits>&
operator<< (std::basic_ostream<cT,
                               traits>& os,
            setbase const& sb) {
  install_bin(os, sb.mBase);
  return os;
}
```

This manipulator is obviously used identically to the `std::setw` or `std::setprecision` manipulators:

```
std::cout << setbase(3) << 10 << "\n";
```

The only problem with this manipulator is that the user can set bases which are out of the supported range (with the code above [2, 16]).

Now let's get back to supporting the standard manipulators: it would be useful if the standard manipulators could still be used, eg. for mixed binary and hexadecimal output:

```
std::cout << "binary: " << bin << i
          << "\n" << "hexadecimal: "
          << std::hex << i << "\n";
```

To do so, the formatting code has to become aware of the use of `std::hex`. This can be detected if the special manipulators clear all bits in the `basefield`: the standard manipulators have to set some bits because the case where no bits are set is treated

specially for integer input (it is equivalent to the %i format specifier of scanf(), i.e. the base of the integer read is determined by the first digits). Thus, the binary formatting code can be rewritten to take special action if the basefield is non-zero. A simple approach is delegating processing to the base class in this case. This is achieved by adding these two lines to the start of the do_put() function:

```
if(fmt.flags() & std::ios_base::basefield)
  return std::num_put<cT,
        OutIt>::do_put(to, fmt, fill, v);
```

The change to the manipulator is even simpler: it just takes the following line to clear the bits in the basefield:

```
ios.unsetf(std::ios_base::basefield);
```

Of course, the overall semantics of using the base class version change the behavior to some extent. At least the open issues noted above are covered. Also, the standard do_put() functions take care of thousands separators (if these are configured for the locale) and some special formatting like upper and lower case letters for hexadecimal values.

Stream Callbacks

As a final round-off to the IOStream manipulator discussion let's deal with those funny callbacks defined in std::ios_base. Streams support registration of callbacks which are called in case of certain events. The main use of these callbacks is support for resource management when associating pointers with streams via the pword() function. There are three events defined in std::ios_base:

erase_event: This event is notified when resources associated with the stream should be released. This event is called when the stream is destroyed and prior to copying when copyfmt() is called.

imbue_event: This event is notified when a new locale is imbue()ed into the stream. Since we modified the locale to take care of binary formatting, the code below demonstrates how this event is caught to modify the new locale, too.

copyfmt_event: This event is notified when copyfmt() is called, after copying all formatting data to the stream. The intent of this event is to either do a deep copy of objects pointed to (the stream merely copies the pointers) or maintain a reference count.

Stream callbacks are rather primitive: only functions with the signature void(*)(std::ios_base::event, std::ios_base&, int) are supported. The first parameter identifies the event being notified, the second identifies the stream object for which the event is notified, and the third parameter is a user parameter passed when registering an event. The callback just handles the imbue_event and imbues a modified locale if the corresponding num_put facet is not a modified one (note that it has to be checked whether the facet is already there to prevent an infinite recursion):

```
template <typename cT, typename traits>
void
bin_callback(std::ios_base::event ev,
             std::ios_base& ios, int) {
  typedef std::ostreambuf_iterator<cT>
                                iterator;
  if(ev == std::ios_base::imbue_event
       && !dynamic_cast<bin_num_put<cT,
                        iterator> const*>(
        &std::use_facet<std::num_put<cT,
             iterator> >(ios.getloc())))
    ios.imbue(std::locale(ios.getloc(),
       new bin_num_put<cT, iterator>()));
}

template <typename cT, typename traits>
std::basic_ios<cT, traits>&
install_bin(std::basic_ios<cT,
            traits>& ios, int base) {
  typedef std::ostreambuf_iterator<cT>
                                iterator;
  if (!dynamic_cast<bin_num_put<cT,
                    iterator> const*>(
        &std::use_facet<std::num_put<cT,
          iterator> >(ios.getloc()))) {
    ios.imbue(std::locale(ios.getloc(),
       new bin_num_put<cT, iterator>()));
    ios.register_callback(
            bin_callback<cT, traits>, 0);
  }
  ios.iword(base_index) = base;
  return ios;
}
```

The callback is registered when a new locale is installed. Since this basically inhibits reinstalling the original locale without using copyfmt() (copyfmt() copies the locale without triggering the imbue_event), it is not necessarily the best design. On the other hand, it might be a reasonable thing to do anyway and the best thing I could think of for demonstrating stream callbacks with this example.

Conclusions

- Manipulators are just functions with certain possible signatures. The possible signatures are

```
std::ios_base& (*)(std::ios_base&)

template <typename cT, typename traits>
std::basic_ios<cT, traits>&
    (*)(std::basic_ios<cT, traits>&)

template <typename cT, typename traits>
std::basic_ostream<cT, traits>&
    (*)(std::basic_ostream<cT, traits>&)

template <typename cT, typename traits>
std::basic_istream<cT, traits>&
    (*)(std::basic_istream<cT, traits>&)
```

- Manipulators can use the functions xalloc(), iword(), and pword() to associate data with a stream.
- Numeric formatting used by the stream classes is done via facets which can be customized to suit specific needs.

Dietmar Kuehl

Ruminations on Knowledge in Software Development

by Allan Kelly

In computing we are accustomed to shunting bits and bytes about. We call this data, we may even accept this represents information, but is it knowledge? In fact, are there any real and important differences between data, information and knowledge? And are these differences of any importance to us when we develop software? (And, with all these questions, am I in danger of turning into a character from a well know HBO series set in New York?)

This article continues the theme of learning from my previous Overload piece, "Software Engineering and Organisational Learning." In part you may like to consider this a simultaneous review of several books which promote the same ideas.

The difference

In everyday language, data, information and knowledge tend to be interchangeable terms. Certainly, most dictionaries I've looked at seem to define each term in terms of the others. However, if there is no difference between these terms what is the point of having them?

For their book, Working Knowledge, Davenport and Prusak (1998) noted that there are many words and definitions that are applied to the nebulous ideas of data, information and knowledge. But since we have enough trouble defining just three terms we had best not ponder on too many. Using their working definitions we get:

- Data claims to be some objective facts about events.
- Information is a message intended to change the receiver's perception of something, it is the receiver rather than the sender who decides what the message means.
- Knowledge is a fluid concept, incorporating experience, values, context that exists inside an individual's mind or in the processes and norms of an organisation.

One of the leading writers on the subject of knowledge is Ikujiro Nonaka, he attributes (1995) three attributes to knowledge:

- Knowledge is about beliefs, commitment, and is a function of perspective and intention
- Knowledge is about action
- Knowledge, and information, are about meaning and are context specific.

Later, he extended these ideas to place knowledge within a concept called "ba" (1998). This is a Japanese term he uses to describe the space in which knowledge exists, take away "ba" from knowledge and what you are left with is mere information.

For example, Meyers's Effective C++ is nothing more than a list of 50 items in strange bizarre language - at least when Nick Hornby publishes a list there are a few laughs. But add experience of C++, the values of the C++ community and the fact that readers are usually practising C++ programmers and suddenly the contents of Effective C++ take on a different meaning.

Another example of "ba" occurred during the development of Concorde. The Soviet Union decided it had to have a supersonic passenger plane to rival the Anglo-French Concorde and the proposed Boeing 2707. Lacking the time and expertise the Soviets stole the blueprints of the plane and set about building their own Koncordski, the Tupulov 144. When revealed the plane looked like Concorde, and it even flew but it didn't perform as expected.

Although they had the plans the Soviet engineers lacked the context and culture of the designs. Measurement systems where different, ways of working were different, and notations were different. Thus, they weren't about to build an exact replica of the Anglo-French plane.

Similar things happen to software project when a new team takes over an old project. The project code may come with documentation and UML charts but it is still difficult to understand. The new team lack the "ba" of the old team. This may explain why developers tasked with maintenance often feel the need to re-write existing code.

Where is knowledge in software development?

The whole software development process is an attempt to codify knowledge. We start with some vague idea of what a system should do and, through successive processes of specification, design, implementation and testing, try to turn that knowledge into a working, useful model.

Our problem is that knowledge is difficult to codify. As software developers our skills and knowledge reside in our own domain, our own field of "ba". We take a problem domain, with its own "ba" field and attempt to produce a product which will exist in both domains, satisfying the requirements of the problem domain while meeting the engineering requirements of our own solution domain.

Software needs to exist simultaneously in these two environments. Commercially it is the part seen by customers that tends to get priority, even though this represents the tip of the iceberg (Figure 1). As engineers we see the bigger, more complex problem underneath the waves.

Codification

As if this weren't enough, much knowledge is actually tacit. That is, it is not codified, it is not written down anywhere. We may not realise we have this knowledge until we attempt to write it down or do things differently. Usually it is just "the way we do it around here."

When we deliver a program it enters into the users' domain. It has to live as part of their "ba" so we must respect what users know and expect. If we embed values and judgements into our software which are different to the ones in common use our customers will find the system counter-intuitive and difficult to use. If, on the other hand, we tailor our system to their norms they will find the system easier to use.

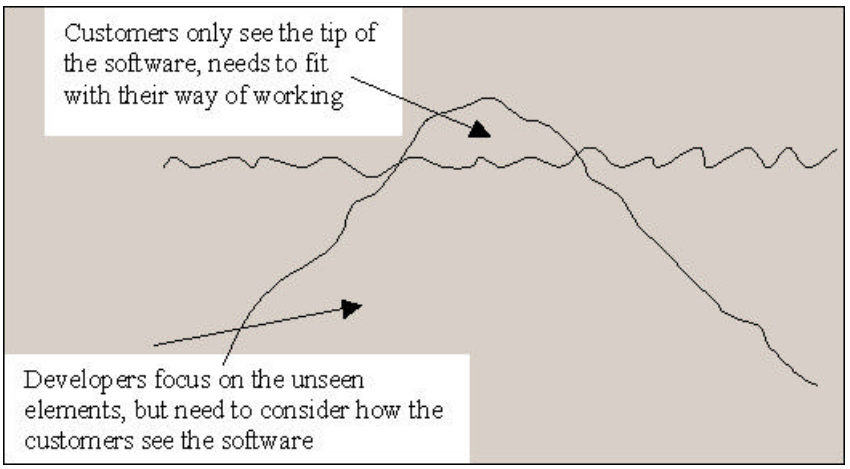

Figure 1 - Software is like an iceberg

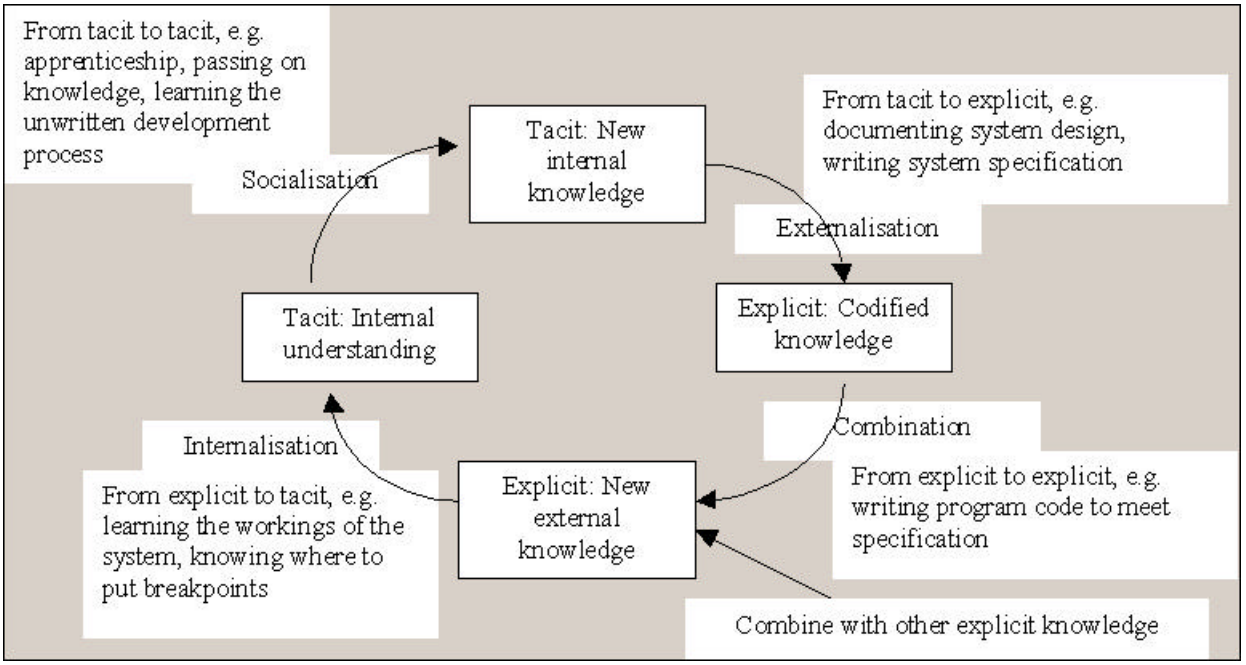

Figure 2 - Nonaka's four modes of knowledge conversion (adapted from Nonaka, 1995, p62

Of course, often the whole point of introducing software is to disrupt current practices so they can be changed. However, we should be sure we know which practices we are attempting to change and which we want to keep. There is no point in introducing software which forces doctors to measure temperatures in Kelvin if we are trying to change their prescribing practices.

Specification

It is when we come to write the specification that we start to grasp the difficulties that are presented by both "ba" and tacit knowledge. Specifications have a tendency to grow like Topsy, they never seem to be complete. If we attempt to write a complete specification we must not only codify the system requirements but also the context, the "ba" they exist in. To be fully complete the specification for the prescribing system would need to explain what temperature is, how it is measured and what the units are.

Specifications are themselves abstractions, and in making the abstractions we have to leave out detail. But the attempt to leave out detail leads to incompleteness because we rely on context to provide it. It is always possible to add more explanation to a specification. Thus we end up with thousand page specifications.

Secondly, our specifications still haven't tackled tacit knowledge. As we write the specification we will uncover more and more undocumented rules of thumb, methods of working, common practices and so on. This continues as the system moves to implementation and we see how the different bits interact. Testing, almost invariably, throws up undocumented assumptions, missed function points and incompatible implementation.

Hand-over

Anyone who has ever worked on a serious software system will have been involved in project hand-overs where a developer attempts to dump the contents of their brain, their knowledge, to a new team member. This can be scary if you're arriving on the team and suddenly trying to absorb a million and one facts about a system, and if you're the one trying to pass on the information - particularly if you're leaving the company.

Documentation is of limited help. Like many developers I've experienced the mountain of documentation which lies in wait when you join a new project. Because it has been written down managers expect that simply reading it will make you as knowledgeable as the writer.

Again we see tacit knowledge and "ba" at work. The documentation can't possibly contain everything the last developer knew about the system. Even if they divided their time equally between documentation and coding there are assumptions that will never make it to paper.

And reading the documentation when you first join a project means you're reading it in the abstract. Until you have been immersed in the project, spoken to other developers - tried to understand the problem and the solution - large parts of documentation are meaningless.

Knowledge creation

In producing a solution to a problem we need to create new knowledge about the process and about the solution. If we understand the knowledge creation process it should help us work with the process rather than against it.

In writing about knowledge, Nonaka, proposes a four stage model (Figure 2) that turns tacit knowledge into explicit knowledge, combines it with other explicit knowledge and turns it back into tacit.

With each conversion knowledge is extended. This may mean it is combined with some other knowledge to create new knowledge, or it may mean that more people understand the knowledge, it may also mean that individuals have a better understanding of the knowledge.

Just do it

Another of Nonaka's points was that knowledge implies action. We need to act on information in order for it to truly be considered as knowledge. After all, how many times have you written a piece of code which you know violates some best practice, but, for whatever reason, time, laziness, expediency, you write it some other way? You have the information to write it better but you choose not to.

Software developers are not alone in this. Newspapers regularly publish stories about reports written for companies or Governments that are not acted on, how a study recommended X in 1998, and in 2001 Y happened because X hadn't been done.

In fact, there is a whole book on subject called - the Knowing Doing Gap by Pfeffer and Sutton (2000). They suggest that individuals, teams, and companies often know what the best thing to do it, but they fail to act on what they know for a variety of reasons. As well as discussing these problems Pfeffer and Sutton examine a number of companies who have succeeded in overcoming these problems and have enjoyed considerable business success.

One of the companies described in Knowing Doing Gap is SAS Institute of North Carolina. SAS is the worlds biggest privately owned software company - proof, if it was needed, that these concepts are applicable to software development.

Perhaps surprisingly Pfeffer and Sutton suggest that successful companies don't have any special secret ingredient, or magic bullet, they don't necessarily do anything other companies don't know about. What these companies do do, is to actually act on what they know. Simple really.

What do we do now?

Many problems in software development are of our own making. We don't do what we know to be right. We use myths to stop us acting on our knowledge, we get involved in infighting and, in many cases, we collude to support a system that we know could be better.

For example, the myth that the 1,000 page specification describes everything that we need to know. No serious software developer really believes this myth but people still contract to develop software on the basis that the specification contains everything we need to know. There is no silver bullet here, the solution is to stop propagating the myth and instead institute working practices that allow for learning and knowledge creation as we go.

Another myth particularly popular among managers is that of the plug compatible programmer - the idea that if a C++ programmer quits we can just hire another C++ trained developer to take their place. I can hear agreement from Overload readers as I write this. However, we developers must bear some of the responsibility here. IT people are known for changing jobs frequently, by doing so we propagate the myth that we can "hit the ground running" and plug a hole quickly.

This myth includes contractors and consultants - the hired guns of the industry. Managers believe they can hire consultants for a short-term role and let them go at a moment's notice. Consultants like this myth because it leads to bigger pay packets and "freedom". But after a while we find managers dependent on contractors and only willing to hire those who have worked in similar roles already. Meanwhile, contractors complain that managers treat them like commodities and don't give them a chance to do something different.

I've been as guilty of this as anybody else. It can be a financially rewarding way to work, and it seems to suit many individuals, and companies like the idea too. However, it leads to an inherent short termism and propagates the plug compatible programmer myth.

In both cases the process and the product are inherently linked. This shouldn't surprise us, processes are created to achieve goals. The problem is that just saying a process is there to achieve "quality" or "on time delivery" does not mean it will. Our processes are far more complex and can produce results we don't desire.

This isn't anything new, this is just another way of stating Conway's law (1968): organisations will produce software which is a copy of its own internal processes. If we want to produce good software, and help our employers succeed, we need to look beyond the immediate issues and see how all the pieces fit together.

Conclusion

Considering software development as learning and knowledge creation highlights the fact that it is difficult to communicate and codify what we want from a piece of software - the old "do what I want, not what I say" syndrome.

[concluded at foot of next page]

The original Conways law

Conway's Law is often applied to software development, a quick search of the web provides references to Wikis specialising in patterns and agile development, Jim Coplien has documented it as an organisation pattern, and it got a few mentions at the ACCU April 2003 conference.

The law is typically quoted something like:

"If there are n developers writing a compiler it will be an n-pass compiler"

"A GUI program developed by x developers will provide x ways of doing the same operation"

"Align architecture with team structure"

The original article is now over 35 years old, but still worth reading. It is quite general in nature giving examples as diverse as transport systems and the US constitution, but does include the compiler example.

Conway builds up his theory with logic, describing how as organisations allocate people to projects they will affect the output of the team. He explains how we can understand communication as a graph with nodes and branches, which will cause the structure of a system to reflect the structure of the organisation that designed it.

The conclusions are still relevant to system designers today:

"The basic thesis of this article is that organizations which design systems (...) are constrained to produce designs which are copies of the communication structures of these organizations. ... a design effort should be structured according to the need for communication."

This causes problems, which need to be addressed by the organisation:

"Because the design which occurs first is almost never the best possible ... flexibility of organization is important to effective design."

As if describing refactoring thirty years before the word was coined wasn't enough he foreshadows by over five years Fred Brooks' Mythical Man Month and what we know as Brooks' Law:

"There is need for a philosophy of system design management which is not based on the assumption that adding manpower simply adds to productivity."

Datamation is no longer published but the short article is well worth reading if you can get a copy of the April 1968 issue.

From Mechanism to Method – Distinctly Qualified

by Kevlin Henney

Introduction

The standard library string type is a product of elegance and sufficiency.

OK, OK, just kidding. If it looks like the product of design by committee it's because it was. If you appreciate minimalist design its baroqueness is certain to disappoint.

The road to Hell is paved with good intentions, and the standard library string is full of good intentions. The standard `basic_string` class template started life as a simple class and reached its middle age as an over-parameterized class template with a bad name and a bloated interface. The process of standardization added somewhat more than two cents worth: What about generalization for internationalization? What about copy optimization through reference counting? What about customizing its memory allocation? What about safe indexing? What about reverse search operations that mirror each forward search operation? What about support for STL? And support for similar index-based operations? You see, good intentions every one of them. But too much compromise in design leads to a compromised design.

In spite of this criticism, I use the standard `string` type. For one thing, it's standard, and for another it satisfies more of my string needs than a vanilla `char *`. More positively, inside this behemoth is a small class (or two) struggling to get out. There is a sense of obligation to try to release and realize it. In this article I don't want to address each and every last detail of such a redesign, but I would like to outline an approach. In particular, I want to declutter the interface a little to clear the path to a better understanding of something that has haunted string classes for the last decade or so: the specter of copy optimization through reference counting and copy-on-write.

The issues thrown up by reference counting can be tackled head on, with limited success, or tackled laterally. The resolution lies in a combination of restraint and substitutability principles [Henney2000], and in particular treating `const` qualification as a form of type separation [Henney2001a, Henney2001b].

Minimalism

The open secrets of good design practice include the importance of knowing what to keep whole, what to combine, what to separate, and what to throw away.

We can start with the name. Names are important. Hiding the templatedness of strings does not actually help the reader in any way, except perhaps to discourage them to think of or use strings as templated abstractions. The reason we are left with the common `typedef`ed names of `string` and `wstring`, and the cumbersome underlying names of `basic_string<char>` and `basic_string<wchar_t>`, is historical. Originally, in the pre-STL era, the proposed standard library was light on templates – in fact it was quite light, period – and `string` and `wstring` were classes. In modern C++ programming we are more familiar with template usage, and would not be quite so reticent about the names or accepting of the supposed benefit of hiding templated usage. If we consider strings to be templated with respect to their character type, then so be it: `string<char>` and `string<wchar_t>` are clearer, more direct, and proud to be templated.

Orthogonality

Designing a string class is more of a challenge than many people appreciate. If you have not already done so (several times) it's worth a try: It's a good C++ work out – memory management, operator overloading, optimization, etc. The problem is not so much in language features or implementation, but in interface. What problem is a string designed to solve? Without a clear focus you discover that everyone has a slightly different view of what a string should be: a self-managed array of characters; a wrapper for `char *` and `<cstring>`; a small piece of text that requires simple access and concatenation; a potentially large stretch of text that requires efficient slicing and rearrangement operations; regular-expression searchable text; and so on. The problem is

---

While software is key to the "information economy" and used by "knowledge workers" we should consider software development itself as knowledge creation. The software development community tends to look inside itself for answers to problems, but there is much we can learn from elsewhere. The writers quoted here aren't specifically interested in software developers but their ideas are highly applicable. Just don't expect technical solutions, these aren't technical problems so there is no technical fix available.

Everything software developers do concerns the application of knowledge and learning. From specification through design to delivery we are concerned with using knowledge and developing products from the application of our existing knowledge and the creation of new knowledge. Understanding this should help improve the development process.

Allan Kelly

Bibliography and further reading

Conway, M. 1968: *How do committees invent?*, Datamation, April 1968

Davenport, T.H., Prusak, L., 2000: *Working Knowledge*, Harvard Business School Press, 2000.

Kolb, D., 1976; "Management and the learning process", *California Management Review*, Spring 1976, Volume 18, Issue 3.

Nonaka, I., Hirotaka, T., 1995: *The Knowledge Creating Company*, Oxford University Press, 1995

Nonaka, I., Konno, N. 1998: "The Concept of 'Ba'", *California Management Review*, Spring 1998, Vol. 40, No. 3

Pfeffer, J., Sutton, R., 2000: *The Knowing-Doing Gap*, Harvard Business School Press, 2000.

WBGH, 1998: *Supersonic Spies*, Nova, transcript at `http://www.pbs.org/wgbh/nova/transcripts/2503supersonic.html`

The UK Channel 4 program Equinox is the US PBS program Nova. In 1998 a programme covered the development of the Tu-144, the Soviet Union's version of Concorde. A transcript of the programme is available from the US PBS site. The UK programme may have been slightly different but the substance was the same.

one of choice: All of these suggestions are reasonable, but satisfying them all simultaneously is not. Any attempt to create a single string class that does so cannot succeed. We have enough existence proofs to back this up.

There is already an excellent example of how to solve this design problem in the standard library: the STL. The STL stands head and shoulders above other container libraries because it is not a container library: It is a specification that defines independent, open-ended families of algorithms, function objects, iterators, and containers along with the requirements that allow them to be combined. Independent ideas are expressed independently, with algorithms separated from underlying container representation through iterators and from functional specifics through function objects. This orthogonal design separates concerns quite clearly. The added bonus is that you get some predefined algorithms, function objects, iterators, and containers thrown into the deal.

There is no single container class that satisfies all of our needs, so we have requirements and exemplars in the library. Given that we now know that asking how to write a single string class is the wrong design question, we can see how the cleaner STL-style solution can be applied to strings. In the first instance, we can consider strings to be sequences. Their bit-copyable elements, common use of null termination, conversions, concatenation, and I/O further characterize them. If we start with this, we end up with a minimal interface that can be satisfied by lightweight `char *` wrappers and scalable string implementations – such as SGI's `rope` template [SGI] – and even `std::basic_string`. The more complex functionality associated with different string types is then expressed orthogonally through algorithms, so that new algorithms can act on all strings and new strings can take advantage of old algorithms. Now, should we want an all-singing, all-dancing, killer-app, last-one-you'll-ever-need string, we can write our own – so long as it satisfies the base requirements of what is means to be a string.

Choice

An interface should represent reasonable goals and present its user with reasonable choices – overachieving interfaces are weaker and more complex, not stronger and simpler.

Consider, for instance, the issue of subscripting. `operator[]` is not required to perform bounds checking whereas `at` is. Indexing out of bounds causes undefined behavior for `operator[]` and an `out_of_range` exception for `at`. On the surface, this looks like a reasonable choice: You get to choose the *quality of failure* for yourself. The problem is that such an option is utterly useless and cannot be reasonably exercised. When would you consciously choose to write code that needed `at` rather than `operator[]`? If you make the choice, you have already anticipated the bug, and can therefore prevent it.

If you have a choice, it should be reasonable to exercise it. It should also be possible to exercise it. Take allocators. Please. The world of containers would be far simpler without them and very, very few people would miss them. People that actually need to customize their memory allocation – for shared memory, for persistence, for the sake of it – find themselves working against rather than with the allocator model. Trying effective memory management of a container whose representation and management is not fully open to you is like eating with a knife and fork... held with chopsticks... through mittens. If you are serious about managing the allocation of a container, then get serious and manage it: Write your own container type. It is simpler, more likely to work, and is very much in the extensible spirit of the STL – more so than limiting yourself to the handful of default container implementations in `std`.

Consistency

Another property of a well-designed interface is consistency. Some functions in `basic_string` throw exceptions on failure, whereas others do not. This is already inconsistent, but is made more so by the presence of both `operator[]` and `at`. If `operator[]` is shadowed by the exception-throwing `at`, then where are the exception-throwing doubles for iteration? If an exception-throwing access operator is considered reasonable, you should expect – indeed, demand – safe and unsafe variants for other forms of access. After all, what is good for the goose is good for the gander. However, we have established that `at` is not reasonable, so there is no need to clutter up the string interface any further.

Mad COW Disease

Strings get copied. Fact of life. Copy assignment and construction afford strings their value-based behavior. But strings are not lightweight classes. They encapsulate a heap allocated representation, and copying could be expensive, especially if the copied string is never modified:

```
template<typename char_type>
class string {
  ...
private:
  ...

  size_t used, reserved; // current
           // length and allocated space

  char_type *text; // allocated and
           // deallocated representation
};
```

The compiler is entitled to a number of optimizations. For instance, the following:

```
std::string cow = "Woof!";
```

Is equivalent to:

```
std::string cow = std::string("Woof!");
```

But can be – and normally is – optimized to:

```
std::string cow("Woof!");
```

For assignment, overloading `operator=` to take a `const char *` prevents a conversion to a temporary that is then used with the ordinary copy assignment operator.

The result of string concatenation is a temporary string object:

```
std::string loud_cow = cow + "!!";
```

Here `operator+` returns a temporary `std::string` object that is used to initialize `loud_cow`. Depending on how the

called function is written, the *named return value* optimization (NRV) allows a compiler to construct directly into `loud_cow` [Ellis+1990, Lippman1996] rather than create an additional temporary object. This optimization applies only to copy construction, not copy assignment: If `loud_cow` is assigned the result of the concatenation, a temporary is created and then discarded. Similarly, in the following initialization two temporaries are created, only one of which can be optimized away by the NRV:

```
std::string loud_cow = cow + " " + cow;
```

Because value objects of class type are commonly passed around by `const` reference, copying typically happens through assignment, data member initialization, and return values. We can see that the compiler already has considerable license to optimize, and that techniques such as overloading to prevent conversions and preferring initialization to construction help reduce the temporary burden, so to speak. But in complex expressions and initialization of data members we can also see that there may still be the need to amortize the cost of copying.

What is required of an optimization? Transparency – so it is substitutable for the unoptimized version – and optimization – many optimizations aren't. In particular, the requirement of transparency means that users should not be entertained by new and interesting bugs.

Counting the Bodies

The most common copy optimization is to share the representation of a string when a copy is made rather than make a deep copy that results in heap allocation. This means that copying is simple and cheap. Only when the string is going to be modified does the 'real' copy occur to avoid aliasing surprises. This lazy, just-in-time model — commonly referred to as *copy on write* — defers the cost of allocation until the point it is absolutely needed. If it is never needed, the cost is not paid. However, few things in life are for free: The sharing is not without overhead. For a start, it must be managed, which increases the complexity of the code. The referencing must also be tracked so that when — as a result of assignment or destruction — a string's text body is no longer referenced it is properly deallocated, and when only a single string handle refers to a text body, redundant deep copies are not made.

There are five ways in which references held by string handles to text bodies can be sensibly tracked, each with its own particular tradeoffs:

1. *Hold separate pointers to the reference count and the actual text*. This means that the footprint of the string object is a little larger and that we are paying for the allocation of two heap objects. The allocation means that it is unlikely that we recoup our investment unless a text body is shared by more than two string handles. For a single reference, this is a not an optimization. Holding a `static` reference count of `1`, and only allocating a dynamic count when the figure rises above that can reduce the overhead in this case. This will complicate the implementation, but if the majority of strings are never copied this will be a saving. If, on the other hand, the string handle's footprint is a concern, the information duplicated between sharing handles can also be associated with the count, reducing the footprint to two pointers:

```
template<typename char_type>
class string {
  ...
private:
  ...
  struct shared {
    size_t used, reserved, count;
  };
  shared *info;
  char_type *text;
};
```

2. *Hold a single pointer to an object that contains the reference count, the pointer to the shared text, and the text size information.* This always results in the allocation of two objects on the heap, and there is an extra level of indirection to reach the actual text. For some designs this could provide an additional benefit of allowing the actual text to be reallocated or virtualized in some way, e.g. to disk, without affecting the handle objects. In the common case, the main benefits of this approach are a little more restricted. The string handle's footprint has now been reduced to a single pointer and, if you want to add a constructor and destructor to the shared body, the management of the text memory can be hidden from the string handle. In its simplest form we can see the basic rearrangement is a proper handle-body configuration [Coplien1992, Gamma+1995]:

```
template<typename char_type>
class string {
  ...
private:
  ...
  struct shared {
    size_t used, reserved, count;
    char_type *text;
  };
  shared *body;
};
```

3. *Hold a single pointer to memory that contains both the information about the string text – including the reference count – and the string text itself.* The information is held as a prefix to the `char_type` array. Only a single pointer is held in the handle, only a single allocation is performed, and treating the space before the text as a different type allows access to the string information. Although this solution is at a slightly lower level, it can be very effective [Henney1998], especially when encapsulated within the string handle. The drawbacks to this approach are that any resize must also involve reallocating and copying the information prefix, and also the intent of the code and connections between the data structures is less obvious:

```
template<typename char_type>
class string {
  ...
private:
  ...
  struct shared {
    size_t used, reserved, count;
  };
  char_type *text; // reinterpret_cast
                   // <shared *>(text) - 1
};
```

4. *Link copied objects together in a doubly-linked list and hold a pointer to the string text.* The information about the string text can be held duplicated in each string handle or as a prefix of the text body's memory. When the links going to the previous and next string handle are both null (or, in a circular configuration, pointing to `this`) the text body is uniquely owned. This style of reference accounting (it is not really reference counting because there is no explicit count) is perhaps least appropriate for strings because there are no operations that require traversal of all handles. Each string handle will have a larger footprint than the other solutions considered so far, although only a single allocation is required per text body:

```
template<typename char_type>
class string {
  ...
private:
  ...
  size_t used, reserved;
  char_type *text;
  string *previous, *next;
};
```

5. *Hold the string text in a managed lookup table and retain some kind of reference into the table.* The information about the string can be held alongside the actual text in the table. This approach is suitable when the aim is not simply to reduce copy cost, but also to eliminate any duplicate strings. It is effectively a symbol table. The cost of initialization from a raw string is increased because of an initial search and a possible initial insertion, and there is increased space overhead per text body that exists. Strings can be held uniquely so that some string features, such as reserved capacity, are no longer appropriate. For strings, the typical implementation is to hold a `static` repository, which introduces its own issues as far as initialization and finalization ordering. This is typically not a suitable design for general purpose strings:

```
template<typename char_type>
class string {
  ...
private:
  ...
  struct less {...}; // function object
type for comparison
  struct info {
    size_t used, count;
  };
  typedef map<const char_type *, info,
less> string_map;
  static string_map strings;
  string_map::iterator entry;
};
```

Clearly, there are many ways to skin a cow. For general-purpose, copy-on-write strings, the first three techniques are the most appropriate and most common.

Trying to be Smart

It seems clear that non-`const` operations such as `operator+=` and `resize` require a string handle to operate on its own copy of the text body. It also seems clear that `const` operations, such as `size` and `compare`, can operate without ill effect on a shared representation. This seems to divide operations in the string world neatly into two type types. However, there is a grey territory in between. What about non-`const` `operator[]`? This operator may be used for both reading from and writing to a string:

```
string<char> cow = "Woof!", ghost = cow;
ghost[3] = cow[1];
```

Both of these calls result in a call to the non-`const` `operator[]`, but for assignment we want to assure that a deep copy happens, but for reading a deep copy would be wasteful. There is no way to distinguish between these uses within `operator[]`. What we need is a smarter reference to do the work for us:

```
template<typename char_type>
class string {
public:
  class reference {
  public:
    ...
    char &operator=(char); // perform
                  // deep copy before write
  operator char() const; // use shared
                        // representation
  private:
    string *target;
    size_t index;
  };
  reference operator[](size_t);
  ...
};
```

This smart reference approach works in most cases. However, a smart reference is not totally substitutable for a real reference. The following fails to compile because `std::swap` expects real references:

```
swap(cow[3], ghost[1]);
```

There are other problems with the smart reference approach for strings [Meyers1996, Sutter1998a], some of which are related to dubious practice – holding the address of a returned reference – and others to do with constraints in the standard – the `reference` type is required to be a real reference, no smart references allowed.

And don't think that the problem is just confined to `operator[]`: It also applies to the `iterator` type, which may be used for both reading and writing. Therefore, for reference-counted strings, `iterator` must be a smart pointer rather than raw pointer type for the reference-counting optimization to be fully effective.

Pessimism

The outlook is pessimistic. As a copy optimization the effectiveness of copy-on-write reference counting has been reduced to a few cases. In other cases it may be quite the opposite of an optimization, regardless of the investment and increase in code complexity.

The only workable evaluation model for these problem functions is a pessimistic one: You don't know whether the user is going to read or write through the returned reference, and you have to just accept that and assume the worst. You may also consider catching some of the corner cases for undefined behavior, such as holding

onto the address of a returned reference. In these cases you have to prevent any future sharing, so that if the current string is used as the source for a copy it causes a deep copy rather than sharing:

```
template<typename char_type>
class string {
public:
  typedef char_type *iterator;
  iterator begin() {
    reserve();
    return text;
  }
  void reserve(); // reserve
representation exclusively
  ...
private:
  ...
  char_type *text;
};
```

All in all, this further reduces the effectiveness of copy optimization to a few corner cases. For non-`const` cases there appears little to be gained from considering this a general-purpose optimization.

Threadbare

The final body blow comes with the introduction of multithreading. Sharing a reference-counted text body becomes unnecessarily interesting when the sharing is between threads. The gut instinct of programmers new to threaded programming is that a mutex or equivalent synchronization primitive will solve the problem. For instance:

```
template<typename char_type>
class string {
  ...
private:
  ...
  struct shared {
    size_t used, reserved, count;
    mutex guard;
    char_type *text;
  };
  shared *body;
};
```

Synchronization primitives are operating system resources, and as such may be potentially scarce and costly to obtain. The temptation is then to share a common mutex for all string objects:

```
template<typename char_type>
class string {
  ...
private:
  ...
  struct shared {
    size_t used, reserved, count;
    char_type *text;
  };
  static mutex guard;
  shared *body;
};
```

In addition to the initialization and finalization issues, you now have another problem: performance. First of all, locking and unlocking mutexes for all data accesses comes with a measurable overhead. And second, all string objects are now serialized through the same mutex, creating a potential bottleneck. Given that the aim of copy-on-write reference counting is to optimize – and taken with all the other issues raised previously – a mutex-based approach is not even on the radar.

If you look carefully at what you need to lock, you will see that the locking revolves around the reference count. Many operating systems provide you with lock-free synchronization primitives for incrementing and decrementing integers, e.g. `InterlockedIncrement` and `InterlockedDecrement` on Win32. With careful coding it is now possible to ensure that no shared text body is ever compromised by race conditions. But note that these primitives still incur a performance penalty – few things in life are free.

Separation of Concerns

There is a question we have to ask ourselves: Is it all worth it? The assumption has always been there that this is a good general-purpose optimization, from the early days of standardization [Teale1991] to the current standard [ISO1998]. At every stage, accommodating this style of implementation has caused headaches, even without the threading issues. The concern is not a recent one [Murray1993]:

> *A use-counted class is more complicated than a non-use-counted equivalent, and all of this horsing around with use counts takes a significant amount of processing time. If the time spent copying values is small enough (either because the values are small and cheap to copy or they are not copied very often), changing the class to do use counting may make programs* slower*. Always do some performance measurements when making this kind of change to convince yourself that this optimization is not really a pessimization!*

With multithreading the issues become even more involved [Sutter1998b] and the horsing around becomes a full-blown stampede (but hopefully not a race condition…). This simply reinforces an earlier conclusion: It is not possible to design a single string implementation that satisfies all uses. Thus the default implementation that causes the fewest surprises (bugs) – either in use or in implementation – is to avoid copy-on-write reference counting. Avoiding it, or providing explicit information on how to disable it, is the approach now adopted by many libraries [Dinkum, SGI].

So deeply rooted is the idea that copy-on-write reference counting is mandatory for strings that many developers are shocked – and sometimes go into denial – when they discover that the return on investment in this technique is often negligible and sometimes negative. The long-standing belief in this old practice is, however, younger than faith in another more fundamental software engineering principle: separation of concerns. And hey, do we have concerns.

A Qualified Difference

Listen to the code, it is trying to tell you something: Mixing reference counting with mutability causes problems. Period. However, if you listen closely, you can hear a leading question, the whisper of a solution: What if you don't mix reference

counting with mutability? What if we are dealing with two related but distinct types?

From an interface perspective, we can see that we can use a string either as something that is read-mostly information or as a read-and-write space. From an implementation perspective, problems with reference counting arise only with mutability. Previously we explored the idea of `const` qualification being a form of subtype relationship [Henney2001a] and one that can be reflected in inheritance [Henney2001b]. For value types we can define separate classes, not related through inheritance, and provide substitutability through conversions [Henney2000].

Consider a design where `string` covers the general case and something like `const_string` covers the immutable case. `const_string` has a subset of the operations of string: the `const` ones plus some that effect a rebinding of handle to text body, such as `operator=`. `const_string` is different to `const string`, which prevents all modification but still comes with any baggage not relevant to `const`, e.g. reserved capacity. It is more like the relationship between `iterator` and `const_iterator`.

Not only do `string` and `const_string` differ in interface, but they can also differ in implementation: `string` should not be reference counted but `const_string` may be. `const_string` has none of the concerns that plagued copy-on-write for a mutable string, and thread safety can be catered for by atomic increment and decrement operations.

Before you get too attached to the names `string` and `const_string` – and assuming that your compiler fully supports partial template specialization – consider one last refinement that uses template specialization and lets us keep a single name:

```
template<typename char_type>
class string {
  ...
private:
  ...
  size_t used, reserved;
  char_type *text; // unshared
};

template<typename char_type>
class string<const char_type> {
  ...
private:
  ...
  struct shared {
    const size_t used;
    size_t count;
  };
  char_type *text; // reinterpret_cast
                   // <shared *>(text) - 1
};
```

With this approach `string<char>` is a common, writeable string and `string<const char>` is the idiom used to work with the read-only variant.

Conclusion

What do you get when cross a string class with copy-on-write reference counting? A problem. What do you get when you cross that with separation according to qualification? A solution.

The road to optimization is full of potholes. Trying to shoe horn many interface and implementation possibilities into a single type leads to twisted back roads. Separating core representation from algorithmic abstraction can clarify and clean up a string interface. Separation according to qualification is also a simplifying decision, both for interface and implementation. A cure – in strings at least – for Mad COW Disease[1].

Kevlin Henney
kevlin@curbralan.com

References

[Coplien1992] James Coplien, *Advanced C++: Programming Styles and Idioms*, Addison-Wesley, 1992.

[Dinkum] *The Dinkum C++ Library*, Dinkumware Ltd, http://www.dinkumware.com/.

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

[Gamma+1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Henney1998] Kevlin Henney, "Counted Body Techniques", *Overload* 25, April 1998, also available from http://www.curbralan.com.

[Henney2000] Kevlin Henney, "From Mechanism to Method: Substitutability", *C++ Report* 12(5), May 2000, also available from http://www.curbralan.com.

[Henney2001a] Kevlin Henney, "From Mechanism to Method: Good Qualifications", *C/C++ Users Journal C++ Experts Forum*, January 2001, http://www.cuj.com/experts/1901/henney.html.

[Henney2001b] Kevlin Henney, "From Mechanism to Method: Total Ellipse", *C/C++ Users Journal C++ Experts Forum*, March 2001, http://www.cuj.com/experts/1903/henney.html.

[ISO1998] *International Standard: Programming Language - C++*, ISO/IEC 14882:1998(E), 1998.

[Lippman1996] Stanley Lippman, *Inside the C++ Object Model*, Addison-Wesley, 1996.

[Meyers1996] Scott Meyers, *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Addison-Wesley, 1996.

[Murray1993] Robert Murray, *C++ Strategies and Tactics*, Addison-Wesley, 1993.

[SGI] SGI Standard Template Library Programmer's Guide, http://www.sgi.com/tech/stl/.

[Sutter1998a] Herb Sutter, "GotW #44: Reference Counting – Part II", September 1998, http://www.gotw.ca/gotw/044.htm.

[Sutter1998b] Herb Sutter, "GotW #45: Reference Counting – Part III", October 1998, http://www.gotw.ca/gotw/045.htm.

[Teale1991] Steve Teale, "Proposing a C++ String Class Standard", *Dr. Dobb's Journal*, October 1991.

This article was originally published on the C/C++ Users Journal C++ Experts Forum in May 2001 at http://www.cuj.com/experts/1905/henney.htm
Thanks to Kevlin for allowing us to reprint it.

1 Thanks to Andrei Alexandrescu's original use of the Mad COW term on comp.std.c++

How To Write A Loop

Jon Jagger

Direct Repetition

```
cout << 1 << endl;
cout << 2 << endl;
cout << 3 << endl;
```

This small code fragment writes the values 1, 2, and 3 to `cout`. You'd be hard pressed to write this code more directly. It writes the value 1, then it writes the value 2, then it writes the value 3. Writing repeated code like this is unrewarding, understanding it is tedious, and modifying it is error-prone and painful. Of course, programmers almost never express repetition like this; they express the repetition more succinctly by adding a level of indirection. Hopefully, by adding a little stylized software you can remove a lot of code elsewhere. Less code, more software. (Note however that in the right context loop unrolling is a useful optimization technique).

Indirect Repetition

```
for (int value = 1; value <= 3; ++value) {
  cout << value << endl;
}
```

This is without question a more succinct way of expressing repetition. If you want to write out the values 1 to 42 simply change the 3 in the continuation condition into 42. However, there is a price to pay for this brevity - the purpose of the code is now expressed less directly. This is the price of indirection. It is totally clear what the purpose of the first code fragment is, to write 1, to write 2, and to write 3. It is not so directly clear what the purpose of the `for` statement is. To understand the `for` statement you have to master the extra complexity: understand `for`'s semantics, mentally examine the initialization, continuation condition, and update parts, and make the correct logical deductions. Experienced programmers know that although the logical deductions required appear trivial and easy, they are in fact fraught with traps and pitfalls. Experienced programmers also know that avoiding mistakes is better than making them, then finding them, and then removing them. Debugging is slow. As a result, programmers tend to learn a few vital mental tools to avoid loop traps and pitfalls. The two most important tools are invariants (things that are always true) and intention (programming on purpose).

Invariants

Here is the unrolled sequence of statements that comprise the previous `for` statement:

```
        initialization: int value = 1;
continuation-condition: (value <= 3); // 1 <= 3, true
                  body: cout << value << endl;
                update: ++value;
continuation-condition: (value <= 3); // 2 <= 3, true
                  body: cout << value << endl;
                update: ++value;
continuation-condition: (value <= 3); // 3 <= 3, true
                  body: cout << value << endl;
                update: ++value;
continuation-condition: (value <= 3); // 4 <= 3, false
```

During this sequence of statements one invariant is "the next value to be written is always inside `value`". That's not a useful invariant because it expresses a truth about the future. A useful invariant expresses a truth about the past, such as "`value-1` is the number of times something has been written". Let's try it.

- Before the loop starts, the invariant says that `(value-1)` writes have already occurred. No writes have occured yet so `(value-1 == 0)`, hence `value == 1`. So `value` is initialized to `1`.
- Then a continuation condition check, and then a write. Now, because a write has occurred, the invariant is momentarily broken: value is still `1`, 1 write has occurred, and `1-1==1` is not true.
- To maintain the invariant you must change `value` to `2` since `(2-1==1)`. The simplest way to change `1` to `2` is via an increment, so that's what the update part does. The update maintains the invariant. And, in general, if N writes have occurred we need to change `value` from N to N+1, which is the definition of 'increment'.
- Now the invariant says that `(2-1==1)` writes have occurred, which is true.

Since the invariant is always true you can, in fact, completely ignore the loop and instead consider the context after the loop. When the loop has finished the continuation condition `(value <= 3)` must be false, so `!(value <= 3)` must be true, viz, `(value > 3)`. Since `value` is initialized to `1`, and is only ever incremented, it must be true that `(value == 4)`. The invariant says that `(4-1==3)` writes have occurred, and three writes have indeed occurred.

!Invariant

So, in fact, often the most important thing about the continuation condition is not the continuation condition itself, but its negation because it's the negation that's true after the loop. Looping is easy; it's knowing when to stop that causes the problems. Consequently you want the negation of the continuation condition to be as strong as possible. In the example, the continuation condition was `(value <= 3)` and hence its negation was `(value > 3)`. In order to strengthen this negation into `(value == 4)` you had to do extra mental work. However, if you weaken the continuation condition you automatically strengthen the negation of the continuation condition. So if instead of writing `(value <= 3)` as your continuation condition you write `(value != 4)` then when the loop finishes you'll know `(value == 4)` with (almost) no mental effort at all. Another benefit of weakening the continuation condition is that it weakens the loop requirements (significant when you consider function templates):

```
for (int value = 1; value != 4; ++value) {
  cout << value << endl;
}
```

At this point it's worth reiterating (sorry) that this code fragment is less direct than the very first code fragment. In the first code fragment the values 1, 2, 3 were written and the values 1, 2, 3 all appeared directly in the code. In this latest code fragment the value 3 does not appear at all. Is this a problem? Does this mean you should use `(value <= 3)` instead of `(value != 4)`? I don't think so. I think it would be a mistake to base any decision on the first code fragment. The reason is simple; programmers don't write code like that. They just don't. Programmers write loops using dedicated loop constructs. That is the context. The secret of programming is not to over generalize or to over specialize; to be aware of, and sensitive to, the immediate problem context.

Dependency

A loop has two parts. The loop control:

```
for (int value = 1; value != 4; ++value)
```

and the loop body being controlled:

```
{
  cout << value << endl;
}
```

These two parts play different roles. The former governs the latter, making sure it executes neither too few nor too many times. There is a clear one-way dependency; the body depends on the control but not vice versa. Any micro-modification that disturbs this dependency is ill advised. For example, suppose you rewrite the `for` statement like this:

```
for (int value = 1; value != 4; ) {
  cout << value++ << endl;
}
```

The loop control now depends on the loop body. In other words the loop control is dependent on the very thing it is supposed to be controlling! Entirely wrong. In fact, the control and the body are becoming intertwined so tightly it's hard to talk about the control as a separate concept at all. The software is disappearing and the loop control and the loop body are gelling into an amorphous lump of code. A lump of code that is less transparent, harder to reason about and harder to understand.

Separation

To avoid this amorphous lump simply don't modify the loop variable inside the loop. That way the dependency remains a one-way dependency from the controlled to the controller, the loop control parts remain separate (and textually together), and the loop invariant remains transparent. As useful as this well-known piece of advice is it's not sufficient to protect your loops. It's not really a generative piece of advice. The most important thing is to keep the loop control separate from the loop body. Separation of Concerns. Modifying your loop variable inside its loop body is one way of breaking the separation and tangling the dependencies but there are plenty of others. Using `goto`, `break`, `continue`, `throw`, or `return` inside the loop body can all have the undesired effect as well. Here's another example where the loop control and the loop body are tightly interwoven. Does it write `1`, `2`, and `3` as before? Are you sure?

```
int value = 1;
for (;;++value) {
  cout << value << endl;
  if (value != 4)
    continue;
  else
    break;
}
```

You might be thinking that advising you not to use `return` statements inside loop bodies is over zealous. Do I really mean that? Yes I do. Functions that return something should do so via a single `return` statement at the very end of the function. Here are some practical reasons why:

- **Change** The only thing absolutely guaranteed in software is change (well, maybe corrupt data too). A function sprinkled with `return` statements will almost certainly break when changed. Such functions are just too opaque. They are not transparent. It's too hard to see, let alone reason about, what effects a change will have. A classic example from C is adding a statement at the start of the function to acquire a resource (eg calling `malloc`) and adding a statement at the end of the function to release the resource (eg calling `free`). Better make sure there's no `return` statement in between.
- **No Change** Software that separates out its concerns and manages the dependencies between the separate parts (in particular what's dependent on what) is significantly easier to refactor than software that does not. If your loop body contains a `return` statement then you won't be able to refactor that loop out into another method. You'll have to first refactor the loop so it doesn't contain any `return` statements.

Half-Open Interval

A quick recap. We started with direct repetition:

```
cout << 1 << endl;
cout << 2 << endl;
cout << 3 << endl;
```

and we've worked through to indirect repetition:

```
for (int value = 1; value != 4; ++value) {
  cout << value << endl;
}
```

The value `3` appears in the direct version but not in the indirect version. As I've already said, I don't think this is worth fretting about since programmers never actually write the first version. However, there is a sense in which `3` does appear, albeit indirectly, in the indirect version. This is because `(1+3==4)` or, equivalently, `(4-1==3)`. Specifying an inclusive lower bound and an exclusive upper bound is an extremely common, powerful, and idiomatic way of expressing a loop. It even has a special notation and name. It's written like this:

```
[1, 4)
```

and it's called a half-open interval. Here are two well-known examples:

```
// [0, 42)
char array[42];
// ...
const size_t end = 42;
for (size_t at = 0; at != end; ++at) {
  stuff(array[at]);
}

// [array, array+42)
char array[42];
// ...
const char * const end = array + 42;
for (char * at = array; at != end; ++at) {
  stuff(*at);
}
```

Note that:

- By definition a legal array index cannot be negative so the first fragment uses a `size_t` to match this constraint (`size_t` is an ISO C/C++ unsigned integer typedef).
- By definition the size of an object fits into a `size_t`. In other words, the valid indexes of the elements of an array of size `N` are `[0, N-1]` but only a `size_t` is guaranteed to be able to hold the value `N`.
- The valid indexes of the elements of an array of size `N` are `[0, N-1]` and their addresses are `[array + 0, array + N - 1]` respectively. However, ISO C/C++ explicitly says you can use the just-past-the-end-address `(array + N)` in pointer comparisons.

Worked Example

The secret of mastering loops (and in fact, of most programming tasks) is to work intentionally. That is, to program on purpose (deliberately) and for a purpose (know what it is you're trying to do).

Suppose your intention is to search through a range of elements looking for a value. A loop is just a mechanism to realize this intention. A loop is, quite literally, a means to an end. If you concentrate on the loop you're solving the solution rather than solving the problem. Concentrate on the problem. Always design a thing by considering it in its next larger context.

If you're searching for an element then either you'll find the element or you won't. You need to be able to distinguish between these two possibilities. One of the strengths of a Half-Open Interval is its exclusive upper bound. When searching:

```
[begin, end)
```

you can make any position (let's call it `at`) in `[begin, end)` correspond to the position of the element if found and make `at == end` correspond to not finding the element. Like this:

```
// ...
if (at == end) // not found
  ...
else           // found
  ...
```

Furthermore, if `(at == end)` is not true it means the element at `at` must equal `value`: `(*at == value)` in the pointer case and `(array[at] == value)` in the indexer case.

```
// ...
if (at == end) // not found
  ...
else {         // found
  assert(*at == value);
  ...
}
```

Now we understand the problem context we can start to think about a solution. If we use a loop then once the loop finishes the following must be true:

```
(at == end) || (*at == value)
```

From the earlier !Invariant section we know that this expression is the negation of the continuation condition. In other words the continuation condition must be:

```
!((at == end) || (*at == value))
```

which, using De Morgans Law, is the same as this:

```
!(at == end) && !(*at == value))
```

which is the same as this:

```
(at != end) && (*at != value)
```

Which means "(we're not at the end) and (we haven't found `value`)". Note how you can't swap the left and right arguments to && because the left side acts a validity check on the right side. It's now just a matter of completing the loop by filling in the initialization part and the update part. Note that you can't declare the loop variable in the initialization part of a `for` statement (since it would be out of scope at the if statement).

```
char array[42];
// ...
const char * const end = array + 42;
char * at = array;
for (; at != end && *at != value; ++at) {
  ;
}
if (at == end) // not found
  ...
else           // found
  ...
```

In the majority of cases finding the value is considered the successful outcome. It's usually best to emphasize the positive case rather than the negative case so a lot of programmers write the if statement like this:

```
// ...
if (at != end) // found
  ...
else           // not found
  ...
```

The empty initialization part and the empty loop body are noticeable. You might be tempted to rewrite the fragment like this:

```
char array[42];
// ...
const char * const end = array + 42;
char * at = array;
while (at != end && *at != value) {
  ++at;
}
if (at != end) // found
  ...
else           // not found
  ...
```

This is possibly a minor improvement. However, a much more relevant point is that to search another array you'd have to write another identically structured loop. Copy-and-paste duplication is a bad thing but it hints at something very important; that you have formed a common pattern of use to conquer similar problems. Instead of copying and pasting you should be considering how to capture and name the common pattern of use in a higher level abstraction. How about a function:

```
char * find(char * begin, char * end, char
value) {
  while (begin != end && *begin != value) {
    ++begin;
  }
  return begin;
}
```

Or a function template:

```
template<typename iterator_type,
         typename value_type>
iterator_type
find(iterator_type begin,
     iterator_type end,
     const value_type & value) {
  while (begin != end && *begin != value) {
    ++begin;
  }
  return begin;
}
```

It's a mistake to think that these abstractions are "too small" to warrant existence. And so finally, we end up with a code fragment that is clear, concise, transparent, and intention revealing:

[concluded at foot of next page]

Embedded Scripting Languages or how to add extra user functionality to your application

by Jonathan Tripp

Why Do It?

What do I mean by an embedded scripting language and why are they useful? By a "scripting language" I mean a simple, cheap (as in free and easy to maintain) and cheerful language with just enough functionality. It should be easy to explain to application users who may have only a little or no programming experience. The syntax should be clear and expressive. It would be better, from the user's perspective, to limit functionality for an easier ride. Think of, for example, an early dialect of BASIC, rather than an object-oriented extension of Lisp. By "embedded", I mean that an interpreter for this language can be integrated into your C/C++ application. This may seem crazy, but it really isn't that difficult and it can be very beneficial.

The principal reason for embedding a scripting language is to allow your application's functionality to be adjusted after it has been built. At the simplest level, most applications choose to externalise some of their operational parameters in a configuration file. This is a reasonable approach if you are able to determine in advance which parameters are likely to change, but that isn't always the case. For example, if your application needs to use a serial port, you could make an entry in a configuration file like:

```
[Comms]
; The port to use
port = "COM2"
timeout = 1000
```

So, a configuration file can be regarded as a group of keyword-value pairs. Here the keyword is `port` with corresponding value COM2. The pairs are grouped into sections separated by the `[Comms]` type line. Optional comments are on lines beginning with a semicolon. Unfortunately, as it stands this isn't always flexible enough, as I shall explain.

I write applications in C++ for controlling scientific equipment. I have a library of routines to control and test each physical component that I combine to build each final application. It is during this stage that I am most exposed to the customer's whims. Much is written about managing projects and customer requirements, but I am not sure that any one system really works. The reality is that a customer may not actually know what they want until they can see a prototype working. For example, in a control environment, suppose you have machines "A" and "B", and specification like:

```
TEST 10 IS:
Switch "A" on
Wait for 5 seconds for it to warm up
Switch "B" on
Wait for 10 seconds for it to warm up
Prime "B"
Trigger "B"
Collect data with "A" for 20 seconds
Switch all off
```

Anticipating that the start-up times will need some fine-tuning, you would externalise them into your configuration file as:

```
[TEST 10]
; A start-up time
Startup_A_Time = 5
; B start-up time
Startup_B_Time = 10
; Collect data for
Collection_Time = 20
```

This works well until someone points out that in fact the instrument start-up order needs reversing. A quick response is to now externalise your "if" clause to the configuration file.

```
; False for reverse start-up order
Startup_A_Then_B = True
```

with corresponding pseudo-code:

```
if (getBoolFromConfigurationFile("Test 10",
                "Startup_A_Then_B") == true) {
  StartA(getIntegerFromConfigurationFile(
                            "Startup_A_Time"));
  StartB(getIntegerFromConfigurationFile(
                            "Startup_B_Time"));
}
else {
  // the other way round
}
```

You can imagine that on a complicated system this is will quickly get silly. Problems like this can and do show up even when installing systems at the client's site. A busy shop floor is definitely not the right environment to be going through the compile/build/link cycle for a large C/C++ application. At this point, what you really want is to be able to program your control algorithm in the configuration file. The installation engineer can then modify the scripts using a simple text editor, reload them into the application and get on with testing. The configuration file must have the ability to present simple functions to your application and call back into your application. So, if we imagine a simple Pascal-like syntax:

```
– A start-up time
Startup_A_Time = 5
– B start-up time
Startup_B_Time = 10
– Collect data for
Collection_Time = 20
– False for reverse start-up order
Startup_A_Then_B = True
```

```
char array[42];
// ...
const char * const end = array + 42;
char * at = find(array, end, value);
if (at != end) {    // found it
  ...
}
```

That's all for now.

The form of this article as well as the content of the first two sections were collectively written by a dozen or so people during a Birds of a Feather session at the ACCU 2003 Spring Conference. Many thanks to everyone who contributed.

Jon Jagger
jon@jaggersoft.com

```
function Test10()
  if (Startup_A_Then_B)
    StartA(Startup_A_Time)
    StartB(Startup_B_Time)
  else
    – the other way round
  end
    –the rest of the algorithm
end
```

This is your chance as the instigator of your fledgling language to make it as simple as possible for non-programmers. Use a simple syntax, i.e., no semi-colons and if possible infer the type from the situation! I think you'll agree that this approach is a lot simpler and easier for non-programmers to understand. It can be argued that the original algorithm is more clearly preserved from the specification. Even more so if you imagine the C/C++ version cluttered with all the ancillary error checking and logging. Its greatest strength is that it is open to change by you, other non-programming engineers and possibly even the end-user. Note that I am not advocating rewriting the whole application in a scripting language, because I consider C/C++ the perfect languages for the controlling libraries.

Examples

I'll now look at some other examples of this technique, in roughly historical order:

Firstly, GNU Emacs. From the Emacs documentation:

Emacs is the extensible, customizable, self-documenting real-time display editor. If this seems to be a bit of a mouthful, an easier explanation is Emacs is a text editor and more. At its core is an interpreter for Emacs Lisp ("elisp", for short), a dialect of the Lisp programming language with extensions to support text editing.

After a few prior implementations, Emacs now consists of a light C core that contains the display code and a Lisp interpreter. The rest of Emacs is programmed in Lisp; the scripts can be edited (in Emacs) and reloaded whilst the system is running. This tremendous flexibility is the main reason why Emacs is loved.

Secondly, for me, are the CAD systems, like AutoCAD. These, like Emacs, generally have a C/C++ core, and also expose a Lisp interpreter. Through Lisp bindings to the core application, the user can write scripts to manipulate much of the system from the graphical user interface to the models.

Thirdly, VBA from the Microsoft Office Suite. From the Microsoft web site:

Finally, Visual Basic for Applications takes the same power available through the Visual Basic programming system and applies it to highly functional applications, enabling infinite levels of automation, customization, and integration.

Since Microsoft's initial business was BASIC interpreters, it should be no surprise that they chose BASIC as the prototype for their embedded language, Visual BASIC for Applications (VBA). Unlike Lisp, small BASIC programs can be written easily with little or no prior programming experience, after all the B is for Beginner's.

Fourthly, computer games: many contemporary games have some form of scripting included. The complexity of a modern game requires it. The core graphics and artificial intelligence libraries are written in C++, but hooks are exposed to an embedded scripting language. Then the script for the game and the levels can be developed, changed and tweaked all in the embedded scripting language. For a popular example, the game "Unreal", developed by Epic Games includes a very sophisticated language called UnrealScript. By exposing this facility they have created a very configurable game engine that can be customised easily. Its versatility is proven by Epic Games selling their engine to other games companies.

Finally, everyone's favourite web server: Apache HTTP Server. This web server also contains a small embedded scripting language for processing what they refer to as "directives". When the server starts, it loads and parses a configuration file, `httpd.conf` by default, which contains directives. These are essentially function callbacks to the main server, with the addition of some conditional processing, based on either command line parameters or module availability.

These are all very successful applications, and I maintain that a large part of their success is due to the fact that they have exposed key configuration data and functions to the end-user.

How To Guide

The simplest way to identify which part of your application would benefit from this is to ask yourself: "which parts of your system are you frequently asked to change?" I think there is a general pattern with most applications; requests for changes will be targeted at those areas the user has most interaction with. This will probably be the gross functionality, i.e. the interactions of your libraries and probably the graphical user interface, if you have one.

In choosing or designing an embedded language, keep in mind your target users. To be accessible for a modern user, you should probably avoid Lisp. I know it is a very powerful language, there are free interpreters available and it is easy to bind to C, but it is a little daunting to a novice. BASIC is fun and like many developers in their 30's it was the first language I learnt on a home computer. You may pause before using Microsoft's VBA since it will require extensive use of COM and it will cost you an indeterminate amount to get a licence from Microsoft. The main scripting languages, Perl, Python and Ruby can all function as an embedded scripting language, and TCL was designed for just such a role. However, I feel they are probably just too inaccessible to a novice. There is a freely available language called Lua that fits my requirements. Lua is available as C source code and comes with a liberal licence. It also has all my other desirables: it is relatively small, can be used as a simple procedural language and has a clean interface to C. Lua was designed to be flexible; it is more of a language framework. It can be coaxed into offering objects with member functions and function overloading, and there are mechanisms available to expose C++ classes directly in Lua. It also uses a virtual machine for speed and performs automatic garbage collection.

First download the Lua source. Version 5.0 has just become available, and I shall be using that. Check you can build it as a static library, and build the standalone Lua interpreter to begin experimenting. This can be used interactively, or alternatively to process a file `test.lua` type `dofile("test.lua")` at the command prompt. Just to get a feel for the language, here is a gentle introduction.

Firstly, note that Lua uses dynamically typed variables. For example:

```
– two global variables
port = "COM2"
timeout = 1000
```

Comments follow two hyphens and continue to the end of the line. `port` and `timeout` are global variables and do not have a type, although their values have types of string and number respectively. Lua has base types of `nil`, `boolean`, `number`, `string`, `function`, `userdata`, `thread`, and

`table`. `nil` is the terminal type, `boolean`, `number` and `string` are all as expected, but note that by default Lua is compiled with numbers as doubles. `function`s in Lua are first class, which means that they can be passed around, created and stored like any other value. `userdata` types are for smoothing integration with C. Lua treats them as simple memory blocks, although this default behaviour can be controlled, as I will show later. `thread`s are new to Lua 5.0 and outside the scope of this article. Finally we have the most important type of `table`, used exhaustively within Lua. A `table` is an associative map, for example:

```
- a global table
default_comms = { port = "COM1",
                  timeout = 5000 }
```

Which creates a global table with keys `port` and `timeout` with corresponding values `COM1` and `5000`. Note that the key may be omitted in which case it defaults to the first unused numeric index:

```
another_comms = { port = "COM1",
                  timeout = 5000, true }
```

will add key `1` with boolean value `true`. The fields can be added or accessed using the familiar dot notation, here using the debugging function `print`:

```
print("Default comms port: ",
default_comms.port, " with timeout ",
default_comms.timeout)
```

will produce the output:

```
Default comms port: COM1 with timeout 5000
```

Lua allows you to define functions:

```
- a global function
function comms_open(port, timeout)
  local time = 1300
  local status = "OK"
  - opening comms port (just fake it for now)
  return time, status
end
- and function call
- this first print will print nils because the
-   time variable has local scope and is now
-   invisible
print("Before comms_open: ", time, status)
time, status = comms_open(another_comms)
print("After comms_open: ", time, status)
```

This is a simple function taking some comms settings and returning the time to start up and a status string to the caller. Since functions are treated like any other value, they can be added to tables. The standard libraries supplied with Lua all package their functions within tables in analogy to namespaces in C++. For example, the standard library for table utilities contains a function `foreach` that can be used as follows:

```
- debugging, print out the contents of a table
-   using the table library foreach function:
table.foreach(another_comms, print)
```

This will visit each of the keys in `another_comms` calling the function `print` with the values `(key, value)`, giving the output:

```
1       true
port    COM1
timeout 5000
```

Lua also supports the usual control structures, as demonstrated by the following function for printing even numbers:

```
– demonstration of control structures
function even_numbers(total)
  local step = 2
  for counter = 0, total, step do
    if counter >= 20 then
      print("Twenties", counter)
    elseif counter >= 10 then
      print("Tens", counter)
    else
      print("Units", counter)
    end
  end
end
even_numbers(24)
```

I think we now know enough for Lua to be a useful language, and we can move directly on to integrating this with the application. To use Lua as an embedded language within a C program, you first need to create an instance of Lua and load in all the standard libraries. Finally this resource should be freed before the program ends with a balancing `close` statement:

```
#include <stdio.h>
#include <string.h>

/* Lua is strictly C, so add a guard for
   C++ compilation */
#ifdef __cplusplus
extern "C" {
#endif

#include <lua.h>
#include <lualib.h>
#include <lauxlib.h>

#ifdef __cplusplus
}
#endif

int main(int argc, char* argv[]) {
  lua_State* L = lua_open(); /* Create a new
                                instance of Lua
                                (lua_State *) */

  /* Initialize Lua standard library
     functions */
  luaopen_base(L);
  luaopen_table(L);
  luaopen_io(L);
  luaopen_string(L);
  luaopen_math(L);
  luaopen_debug(L);

  /* do some stuff */

  lua_close(L);
  return 0;
}
```

All communication between C and Lua is done using a stack mechanism, function call parameters are pushed, the call is made and the results will be on the stack. Positive stack indices are from the bottom and negative stack indices are from the top, as usual a `push` adds elements to the top of the stack. So, to add a new global variable and a new global table to Lua:

```
/* Create a global variable in Lua */
lua_pushstring(L, "baud_rate");
      /* push the variable name */
lua_pushnumber(L, 9600);
      /* then its value */
lua_settable(L, LUA_GLOBALSINDEX);
      /* finally set it in the global table */

/* We can create a global table too */
lua_pushstring(L, "backup_comms");
      /* push the table name */
lua_newtable(L);
      /* create a new table on the stack */
lua_pushstring(L, "timeout");
      /* push the field name and value */
lua_pushnumber(L, 2500);
lua_settable(L, -3);
      /* now the table we created has been
         pushed to -3 */
lua_settable(L, LUA_GLOBALSINDEX);
      /* finally set it in the global table */
```

The functions `lua_push***(L, ***)` just push their datatypes onto the stack. The function `settable` adds the field `baud_rate` with value `9600` to the table at the stack index `LUA_GLOBALSINDEX`. This is a special reserved index to identify the table that holds the global variables. This C code is directly equivalent to the following Lua code:

```
baud_rate = 9600
backup_comms = { timeout = 2500 }
```

Note that it is possible to get Lua to execute code fragments from C as follows:

```
lua_dostring(L, "baud_rate =
                 9600\nbackup_comms =
                 { timeout = 2500 }");
```

To make a C function callable from Lua we follow the stack conventions above. Note that when Lua calls C it does so with a clean stack each time. The calling parameters are available in stack indices +1, +2 etc, and on return push the return values. For example:

```
/* A C function callable from Lua */
int l_comms_open(lua_State *L) {
  const char *port = NULL;
  double timeout = 0.0;
  double time = 1300;
  const char *status = "OK";
  /* Function parameters passed in at the
     beginning of the stack */
  if (lua_isstring(L, 1))
      /* check that the first parameter is
         a string */
    port = lua_tostring(L, 1);
      /* retrieve the first parameter */
  if (lua_isnumber(L, 2))
      /* ditto for numbers */
    timeout = lua_tonumber(L, 2);

  /* Do something interesting...omitted */

  lua_pushnumber(L, time);
      /* push the return values */
  lua_pushstring(L, status);
  return 2;
      /* return the number of results */
}
```

This function can be registered in C as a global function in Lua as follows:

```
lua_register(L, "c_comms_open",
             l_comms_open);
```

Now we have passed some data down to Lua, we can load a Lua script and see how it all works together. Add the following to the end of our test script:

```
print("baud_rate: ", baud_rate)
table.foreach(backup_comms, print)
time, status = c_comms_open("COM4",
                        backup_comms.timeout);
print("Result of comms_open: ", time, status)
```

To load a script from C and confirm the variables are populated correctly, use:

```
lua_dofile(L, "test1.lua");
```

We can also get values from Lua and invoke functions in Lua in the same manner. To get a global value, just push the table key and call `lua_gettable(L, LUA_GLOBALSINDEX)` to ask Lua to look up the value and put it at the top of the stack. Similarly, to get a value from a global table, first ask Lua to lookup the table and put it on the stack, and then push the table key before calling `lua_gettable` to finally lookup the value. Using the same test script we can make a call to the Lua `comms_open` function as follows:

```
/* Call a Lua function */
lua_pushstring(L, "comms_open");
      /* ask Lua to find the global function
         and push it onto the stack */
lua_gettable(L, LUA_GLOBALSINDEX);
if (lua_isfunction(L, -1)) {
  lua_pushstring(L, "COM4");
      /* push the two operands */
  lua_pushnumber(L, 3500);
  lua_call(L, 2, 2);
      /* make the function call, two inputs
         and two outputs */
  if (lua_isnumber(L, -2))
      /* results will be on the top of the
         stack */
    time = lua_tonumber(L, -2);
  if (lua_isstring(L, -1))
    status = lua_tostring(L, -1);
  lua_pop(L, 2);
}
```

At this point we can get and set Lua global variables from C, and call Lua global functions from C. We can also callback from Lua into C and load and execute Lua scripts. The example functions above could easily be isolated into a Lua interface library, and I think there is an obvious wrapping into a C++ class if you'd prefer. This is enough functionality to begin exploring Lua and C integration in earnest. As I mentioned earlier the userdata type available in Lua and hinted that its behaviour could be modified. In fact, Lua exposes most of its internal functionality through "metatables" and these can be modified from either C or Lua script itself. These tables are used to hold the operators for each object. To take the example from the Lua documentation, consider the behaviour of adding two objects. The internal processing done by Lua is as follows: if both operands are numeric, just add them together. Otherwise, if the first operand has an `__add` field in its metatable, use that function, otherwise consider the second operand's metatable. The operators available for overloading in this way are: `__add`, `__sub`, `__mul`, `__div`, `__pow`, `__unm` (for unary minus), `__concat` (for string concatenation), `__eq`, `__lt` (less than) and `__le` (less than or equal), `__index` (for field getters) and `__newindex` (for field setters) and `__call` (for function calls). Additionally for userdata types there is the `__gc` event called by the garbage collector for object finalisation. To see how this can be used for your userdata types consider the following example:

```
#define COMMSHANDLE "Comms*"

typedef struct tagComms {
  char *port;
  double timeout;
} Comms;

static int l_comms_new(lua_State *L) {

  /* Create a new userdata object of the
     correct size */
  Comms *comms =
      (Comms *)lua_newuserdata(L,
                               sizeof(Comms));
  comms->port = NULL;
  comms->timeout = 0.0;

  return 1;
}
```

If you now register this function with Lua then when it is called it will create a new Comms struct and initialise it. Unfortunately, since this example does not contain just plain old data, there will be a memory leak for each of these structures since there is no way to clean them up. To remedy this, we need to create a new metatable object implementing the correct garbage collection to override the default Lua behaviour for userdata types. To create a new metatable object, just use the function:

```
luaL_newmetatable(L, COMMSHANDLE);
  /* create new metatable for file handles */
```

and to attach the Comms userdata objects to this metatable, just add the lines

```
luaL_getmetatable(L, COMMSHANDLE);
    /* retrieve the metatable for this type */
lua_setmetatable(L, -2);
    /* set this metatable for this object */
```

to the Comms constructor function above. Connecting a userdata object with its metatable in this way is the Lua equivalent of constructing a v-table for a C++ object with virtual member functions. You can override the garbage collection behaviour by creating a C function as follows:

```
/* Define the garbage collection
   finalizer for Comms objects */
static int l_comms_gc(lua_State *L) {
  Comms *comms =
              (Comms *)lua_touserdata(L, 1);
  free(comms->port);
  comms->port = NULL;
  return 0;
}
```

Register this as the garbage collection routine for this metatable as follows (assuming that the metatable object is currently on the top of the stack)

```
lua_pushliteral(L, "__gc");
lua_pushcfunction(L, l_comms_gc);
lua_settable(L, -3);
```

I hope I have shown that your application could benefit from an embedded scripting language of some form. I have discussed some of the prior examples and have introduced a more modern language called Lua. I've given a quick taste of Lua and indicated that it can be extended easily and it can be embedded easily. The interface between C and Lua is easy to understand and easy to isolate. There are examples of other third party wrappers available that promise to even wrap C++ classes for easy access from Lua. There is also a growing body of third party libraries available for processing XML and for creating GUIs with Tk.

Jonathan Tripp

References

[1] Emacs: `http://www.gnu.org/software/emacs`
[2] VBA: `http://msdn.microsoft.com/vba/default.asp`
[3] TCL: `http://www.scriptics.com/advocacy/tclHistory.html`
[4] John K. Ousterhout: `http://home.pacbell.net/ouster/scripting.html`
[5] Lua: `http://www.lua.org/`
[6] Unreal Script: `http://unreal.epicgames.com/UnrealScript.htm`
[7] Apache: `http://httpd.apache.org`