Generic Programming in C++: A modest example

Marshall Clow
Qualcomm, Inc.
mclow@qualcomm.com
marshall@idio.com

C++Now
May 2012

Problem Definition

On November 1st, on the Boost developers list, Olaf asked:
> **Does Boost have hex/unhex() that support std::string?**
> **I can't find them and I think they'd be quite handy to have.**

His Proposed Interface

```
std::string hex   ( const std::string &input );
std::string unhex ( const std::string &input );
```

That's not very generic

- CString
- QtString
- Other string classes
- What about other containers?

What about Unicode?

- wchar_t – 16 bits on Windows, 32 on Unix
- C++11 defines char16_t and char32_t
- Different string types for them, too

My Proposed Interface

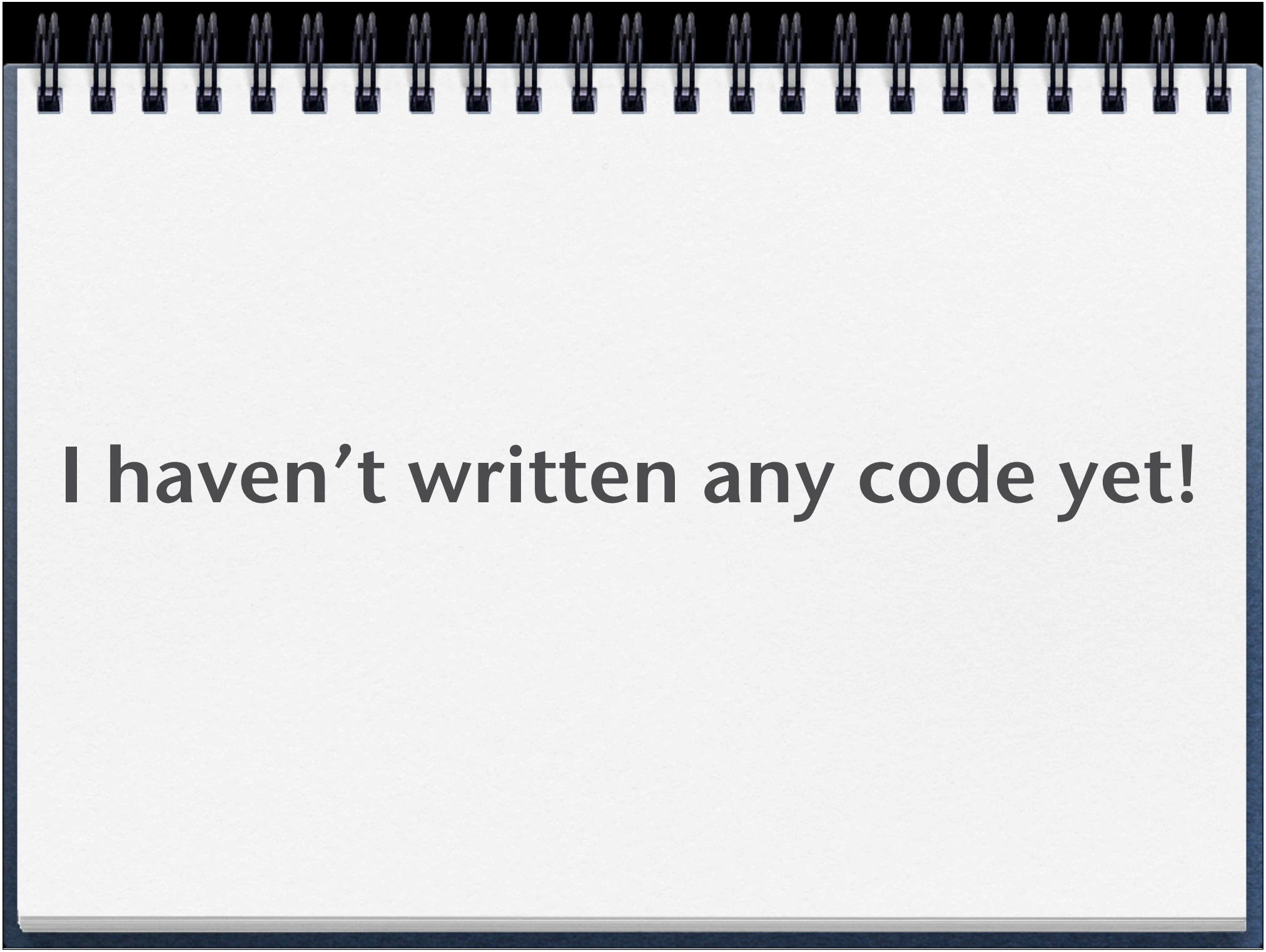
```
template <typename InputIter, typename OutputIter>
OutputIter hex   ( InputIter first,
      InputIter last, OutputIter out );

template <typename InputIter, typename OutputIter>
OutputIter unhex ( InputIter first,
      InputIter last, OutputIter out );
```

What about string literals?

```
template <typename OutputIter>
OutputIter hex   ( const char *p, OutputIter out );
```

And Boost.Range, too:

```
template <typename Range, typename OutputIter>
OutputIter unhex ( const Range &r, OutputIter out );
```


I haven’t written any code yet!

First building block

```
template <typename T, typename OutputIterator>
OutputIterator encode_one ( T val, OutputIterator out ) {
  const std::size_t num_hex_digits = 2 * sizeof ( T );

  char res [ num_hex_digits ];
  char  *p = res + num_hex_digits;
  for ( std::size_t i = 0; i < num_hex_digits; ++i, val >>= 4 )
    *--p = "0123456789ABCDEF" [ val & 0x0F ];
  return std::copy ( res, res + num_hex_digits, out );
  }
```

What operations are required on T?

- copy constructor
- sizeof
- operator >>= (right shift)
- operator & (bitwise AND)

How do we figure out what T is? (The compiler knows!)

```
template <typename InputIterator, typename OutputIterator>
OutputIterator hex ( InputIterator first, InputIterator last,
            OutputIterator out )
{
    while ( first != last )
        out = encode_one (*first++, out);
    return out;
}
```

Other Versions

```
template <typename T, typename OutputIterator>
OutputIterator hex ( const T *ptr, OutputIterator out )
{
     while ( *ptr )
        out = encode_one (*ptr++, out);
     return out;
}

template <typename Range, typename OutputIterator>
OutputIterator> hex ( const Range &r, OutputIterator out )
{
    return hex (boost::begin(r), boost::end(r), out);
}
```

What about decoding?

- Decoding can fail
  - Bad input (non-hex “characters”)
  - Not enough input
- What should happen in those cases?

First building block

```
unsigned hex_char_to_int ( char c ) {
    if (c >= '0' && c <= '9') return c -'0';
    if (c >= 'A' && c <= 'F') return c -'A'+10;
    if (c >= 'a' && c <= 'f') return c -'a'+10;
    throw std::runtime_error ( "Non-hex char");
    return 0; // keep dumb compilers happy
    }
```

Second building block

```
template <typename InputIter, typename OutputIter>
OutputIterator decode_one ( InputIter &first, InputIter last, OutputIter out )
{
  typedef typename std::iterator_traits<OutputIter>::value_type T;
  T res(0);

  // Need to make sure that we get can read that many chars here.
  for ( std::size_t i = 0; i < 2 * sizeof ( T ); ++i, ++first ) {
    if ( first == last )
      throw std::runtime_error ( "Not enough input" );
    res = ( 16 * res ) + hex_char_to_int (static_cast<char> (*first));
    }

  *out++ = res;
  return out;
}
```

What operations are required on T (for decoding)?

- copy constructor
- sizeof
- assignment from int (zero)
- operator * (multiplication) (could use <<)
- operator + (addition) (could use '|')

And we're almost home!

```
template <typename InputIter, typename OutputIter>
OutputIter unhex ( InputIter first, InputIter last,
                   OutputIter out )
{
    while ( first != last )
       out = decode_one ( first, last, out );
    return out;
}
```

Pointer based version
What should XXX be?

```
template <typename T, typename OutputIter>
OutputIterator unhex (const T *ptr, OutputIter out)
{
    while ( *ptr )
       out = decode_one ( ptr, XXX, out );
    return out;
}
```

Let's write some tests!

```
#include <iostream>
#include "hex.hpp"

int main ( int argc, char *argv[] )
{
      for ( int i = 1; i < argc; ++i ) {
          std::string hstr, res;
          hex   ( argv[i], std::back_inserter ( hstr )); // pointer-based version
          unhex ( hstr,    std::back_inserter ( res ));  // range-based version
          if ( res != argv [i] ) {
              std::cerr << "# Round Trip failed!" << std::endl;
              std::cerr << "   " << argv[i] << std::endl;
              std::cerr << "   " << hstr << std::endl;
              std::cerr << "   " << res << std::endl;
              }
          }
      return 0;
}
```

It did not compile!

- The upshot of the compiler errors was “you can’t declare a variable of type ‘void’”
  - The offending line was:
  - `T res (0);`
- Turns out that output iterators have a value_type of ‘void’

Where do we go from here?

- Anger – This stupid s*#@! is broken!
- Denial – Why don't output iterators have a value_type?
  - Research the history
  - The answers I found are unsatisfactory
- Acceptance – the standard is not going to change.

Working around the problem

- I decided that I didn't need to support all output iterators.
- I definitely wanted to support back_inserter_iterators.
- ostream_iterators would be nice, too.

“Reaching inside” the iterator

```
// The general case
template <typename Iterator>
struct hex_iterator_traits {
  typedef typename
      std::iterator_traits<Iterator>::value_type value_type;
};

// Specific for back_insert_iterators
template<typename Container>
struct hex_iterator_traits<std::back_insert_iterator<Container> >
{
     typedef typename Container::value_type value_type;
};
```

Packaging it up

```
template <typename Iterator>
struct iterator_value_type
{
  typedef typename
    hex_iterator_traits<Iterator>::value_type value_type;
};
```

Then I changed the line in decode_one:

```
typedef typename iterator_value_type<OutputIter>::value_type T;
```

And it worked!

- And when I tried it with std::wstring, it worked, too!
- And with std::vector<char>
- And with std::list<char>
- And with std::deque<unsigned long>
- with different output iterator types, too!

Polish and fit

- Hiding the messy details
  - namespace ‘detail’
- Prefer pre-increment to post-increment
- Use Boost.Exception
- Restricting to integral types
  - Using boost::enable_if
- Test cases

Using Boost.Exception

- Declared a (very small) exception hierarchy
- `non_hex_input` carries the offending char
- Changed the throwing code to look like:
  - `BOOST_THROW_EXCEPTION (non_hex_input (c))`
- Updated tests to catch new exception types.

Boost.Exception

```
struct hex_decode_error:
  virtual boost::exception, virtual std::exception {};

struct not_enough_input : public hex_decode_error {};

struct non_hex_input : public hex_decode_error {
     non_hex_input ( char ch ) : bad_char ( ch ) {}
     char bad_char;
private:
     non_hex_input ();      // don't allow creation w/o a char
     };
```

enable_if

```
template <typename InputIter, typename OutputIter>
typename boost::enable_if<boost::is_integral<
  typename detail::hex_iterator_traits<InputIter>::value_type>,
  OutputIter>::type
hex ( InputIter first, InputIter last, OutputIter out ) {
     for ( ; first != last; ++first )
        out = detail::encode_one ( *first, out );
     return out;
     }
```

Circling back around...

```
template<typename String>
String hex ( const String &input ) {
     String output;
     output.reserve (
        input.size() * (2*sizeof (typename String::value_type)));
     (void) hex (input, std::back_inserter (output));
     return output;
     }

template<typename String>
String unhex ( const String &input ) {
     String output;
     output.reserve (
        input.size() / (2*sizeof (typename String::value_type)));
     (void) unhex (input, std::back_inserter (output));
     return output;
     }
```

Alternatives

At one point in the discussion on the Boost developers' list, I opined that these routines were probably a one-liner in Boost.Spirit. michi7x7 <michi@michi7x7.at> took up the implied challenge, and implemented the routines using spirit.

It turns out that I was wrong - they're a few lines.

Performance

- Performance comparable or better than non-template code
  - Interesting performance issues with std::string
- Be careful to compare apples to apples

Summary

- 1 header file
  - 6 public templates
  - other code “hidden” in detail namespace
  - about 130 lines of (non-blank, non comment) code.

Questions?

- Code is in “boost/algorithm/hex.hpp”
- Will be in Boost release 1.50
- Questions to marshall@idio.com


Thank you!