Explore

Printing containers

Danny Havenith
Jared McIntyre

Big Disclaimer

- This talk will be more about what's **not** in explore.
- ...and why

Contents

- Boost library in a week '07
- Library requirements
- Design history
- What's not in Explore
- How to use explore
- Code walkthrough
- Discussion

LIAW'07

- Morning sessions
- Attended by ~25 persons
- During BoostCon mostly feasibility/requirements
  - Lots of inputs from many of the attendees
  - Complex formatting requirements
- Real development after BoostCon
  - 3 persons coding/testing
  - ...and one manager
  - simplified library
- Need for feedback (this session)

Requirements

Use Case: write my container (debugging/logging)

```
template< typename FwdIter>
std::ostream &stream_range( std::ostream &o, FwdIter &b, FwdIter &e)
{
    if (b != e)
    {
        o << *b++;
        while (b != e)
        {
            o << ',';
            o << *b++;
        }
    }
    return o;
}

    template<typename Elem, typename Tr, typename T, typename Allocator>
    std::basic_ostream<Elem, Tr>& operator<<(
                std::basic_ostream<Elem, Tr>& ostr,
                const std::vector<T, Allocator>& v)
    {
        return stream_range( ostr, v.begin(), v.end());
    }
```

Done!

Requirements (ctd)

- Multi-level formatting
- Customizable formatting
- Allow default formats
- Custom containers

```
|1,2,3|
|4,5,6|
|7,8,9|
```

```
<table>
<tr><td>1</td>...</tr>
...
</table>
```

!Requirements

- Decision: concentrate on (start, separator, end) formatting, no fancy karma stuff
- No input formatting
- Just containers, not the kitchen sink

(1, 2, 3)
< red:green:blue>

Design History

- Investigated several implementations
  - straight operator<<, formatting in stream state
  - print(...) - free function with formatter argument
  - boost.format-like operator%, formatter argument
- Implemented print (Danny) and operator<< (Jeff F)
- Removed print again
  - Interference with operator<<
  - KISS

Supported Containers

- Simple Containers
  - c-style array
  - std::deque
  - std::list
  - std::pair
  - std::set
  - std::multiset
  - std::vector
  - boost::array
  - boost::range
  - boost::tuple
- Associative Containers
  - std::map
  - std::multimap
- Custom Containers
  - requires custom operator <<

Simple Containers

```
#include <boost/explore.hpp>
using namespace boost::explore;

// vector of int
std::vector<int> vi;
vi.push_back(1);
vi.push_back(2);
vi.push_back(3);
std::cout << vi;

// vector of vector of int
std::vector<std::vector<int> > vvi;
vvi.push_back(vi);
vvi.push_back(vi);
std::cout << vvi;

// c-style array of int
int arr[3] = {1,2,3};
std::cout << arr;
```

Output

```
[1, 2, 3]

[[1, 2, 3], [1, 2, 3]]

[1, 2, 3]
```

Associative Containers

associative containers are not containers of pairs

```
std::map<int,std::string> mis;
mis.insert(std::make_pair(1, “first”));
mis.insert(std::make_pair(2, “second”));
mis.insert(std::make_pair(3, “third”));
std::cout << mis;
```

Output

~~[[1, first], [2, second], [3, third]]~~

[1:first, 2:second, 3:third]

lexical_cast

```
vector<int> vi;
vi.push_back(1);
vi.push_back(2);
vi.push_back(3);
string outVal( lexical_cast<string>(vi) );
```

Result

outVal == [1, 2, 3]

Custom Delimiters

Manipulator	Default	Description
start(char*)	'['	Changes the string output at the beginning of a container.
end(char*)	']'	Changes the string output at the end of a container.
seperator(char*)	', '	Changes the string output in between elements.
assoc_start(char*)	''	Changes the string output at the front of an association.
assoc_end(char*)	''	Changes the string output at the end of an association.
assoc_seperator(char*)	':'	Changes the string output in between elements of an association.

```
std::vector<int> vi;
vvi.push_back(1);
vvi.push_back(2);
vvi.push_back(3);
std::cout << start("<") << end(">") << vi;
```

Output

~~[1, 2, 3]~~

<1, 2, 3>

Leveled Delimiters

```
std::ostream& format_2d(std::ostream& ostr)
{
    // level 0
    ostr << start("") << end("") << separator("\n");
    // level 1
    ostr << start("|", 1) << end("|", 1) << separator(" ", 1);
    return ostr;
}
```

```
std::vector<int> vi;
vi.push_back(1);
vi.push_back(2);
vi.push_back(3);
std::vector<std::vector<int> > vvi;
vvi.push_back(vi);
vvi.push_back(vi);
vvi.push_back(vi);

std::cout << format_2d << vvi;
```

Output

~~[[1, 2, 3], [1, 2, 3], [1, 2, 3]]~~

```
|1 2 3|
|1 2 3|
|1 2 3|
```

Stream Custom Containers

```
class user_vector
{
public:
   user_vector()
   {
      m_vec.push_back(1);
      m_vec.push_back(2);
      m_vec.push_back(3);
   }

   std::vector<int>::const_iterator begin() const
   {
      return m_vec.begin();
   }

   std::vector<int>::const_iterator end() const
   {
      return m_vec.end();
   }

private:
   std::vector<int> m_vec;
};
```

Stream Custom Containers

```
std::ostream& operator<<(std::ostream& ostr, const user_vector& u)
{
   return explore::stream_container(ostr, u.begin(), u.end());
}
```

```
user_vector v;
std::cout << v;
```

Output

```
[1, 2, 3]
```

For associative containers that use pair-conforming objects:

```
std::ostream& operator<<(std::ostream& ostr, const user_vector& u)
{
   return explore::stream_container(ostr, u.begin(), u.end(),
                                    explore::stream_associative_value());
}
```

Code Walkthrough

- Overview follows, not everything is shown here

Code Walkthrough (1/4)

```
#include <boost/functional/detail/container_fwd.hpp>
namespace std
{
    // stream vector<T>
    template<typename El, typename Tr, typename T, typename Allocator>
    std::basic_ostream<El, Tr>& operator<<(
                std::basic_ostream<El, Tr>& ostr,
                const std::vector<T, Allocator>& v
            )
    {
        return explore::stream_container(ostr, v.begin(), v.end());
    }
}
```

Code Walkthrough (2/4)

```
template<...>
std::basic_ostream<Elem, Tr>& stream_container(
        std::basic_ostream<Elem, Tr>& ostr, FwdIter first, FwdIter last, F f)
{
    // ...stuff omitted...
    // starting delimiter
    ostr << state->start(depth);
    while( first != last )
    {
        // value
        f(ostr, *first, state);
        if( ++first != last )
        {
            // separation delimiter
            ostr << state->separator(depth);
            // ...stuff omitted...
        }
    }
    // ending delimiter
    return ostr << state->end(depth);
}
```

Code Walkthrough (3/4)

```
template<typename T>
int get_stream_state_index()
{
    static int index = std::ios_base::xalloc();
    return index;
}
```

Code Walkthrough (4/4)

```
template<typename T>
T* get_stream_state(std::ios_base& stream, bool create)
{
    // grab reserved index
    int index = detail::get_stream_state_index<T>();

    // grab state data at that index, casting from void*
    T*& state = reinterpret_cast<T*&>(stream.pword(index));
    if( !state && create )
    {
        // both creating a new T and registering the callback allocate memory.  Use
        // auto_ptr to satisfy the strong exception guarantee.
        std::auto_ptr<T> pt(new T);
        stream.register_callback(detail::delete_extra_state<T>, index);
        state = pt.release();
    }
    return state;
}
```

Summary

- Lets keep container printing simple
  - Based on operator<<
- Begin-delimiter-end formatting can get you quite far
- stream_state to wrap xalloc, pword, etc
- operator<< for T needs to be in namespace of T
  - Consequence: in std for std containers!

Discussion

Remarks from the audience

! mark strings with quotes
! having escaping for strings
! separate associative begin-end.
! retrieve sticky state. Create guard to store formatting state.
! Should 0 always mean current level. It should, it doesn't yet.
what happens when you pass the last number of depth defined? stick with last or modular?
start-end as a pair.
function callback manipulators
check whether adding functions in std has a precedence in serialization
is_streamable metafunction: wrap to get rid of assumption of sizeof(s << 0) != sizeof(char)
Is this different than serialization? Can you do the same thing with it? Answer: it's not the same, it is output only and the output is intended for human consumption.

explore_print

- “print” instead of “<<” to avoid ambiguous function overload-errors
- print(x, my_stream) should always compile
  - And do something useful...
- print uses *operator*<< if available...

explore_print

- Does type T have an *operator*<< for streams?

```
struct AlmostAnything
{
    template<typename T> AlmostAnything( T const &);
};

// operator<< cannot be variadic (“have '...' as argument”), so we add this
// one to the overload set, which should have quite low priority.
char operator<<( std::ostream &, AlmostAnything const &);
template< typename T> T& make_t();

template< typename T>
struct is_streamable : bool_< sizeof( std::cout << make_t<T>()) != sizeof( char)>
{
};
```