A Tour of Two New Boost Libraries

ACCU 2005 Conference, Oxford

Thorsten Ottosen (nesotto@cs.aau.dk)

Department of Computer Science
Aalborg University

26th April 2005

By the way...

```
--------------------------
CodeProject Poll
--------------------------
WEEKLY POLL RESULTS
Do Developers have a Life?
http://www.codeproject.com/script/survey/detail.asp?survey=372

 I have an excellent social life       362  20.6
 I have an OK social life              627  35.7
 I have a pretty poor social life      457  26.0
 I have no life. None.                 310  17.7

 Responses                                  1756
```

Overview

Motivation (1)

If syntactic sugar didn't count, we'd all be programming in assembly language.

- Generic algorithms are cool ... but
- Using standard algorithms is clumsy

```
vector<int> some_vector;
...
sort( some_vector.begin(), some_vector.end() );
```

- The syntax is not uniform

```
pair<iterator,iterator> p = ... ;
...
sort( p.first, p.second );
```

Motivation (2)

- Arrays are clumsy

```
#define ARRAY_SIZE( a ) (sizeof(a)/sizeof(a[0]))
int array[size];
...
sort( array, array + ARRAY_SIZE( array ) );
```

- ... and tricky

```
char array[] = "some data";
...
sort( array, array + ARRAY_SIZE( array ) - 1 );
```

- ... and what about `char*`?

Motivation (3)

- And what about iterators?
- This only works for containers

```
template< class Container >
class Foo
{
    typedef typename Container::iterator;
...
```

- What is the iterator type of an array?
- What about the other essential types like `const_iterator` and `value_type`?

The Solution

The Fundamental Theorem of Software Engineering (B. Lampson):

> *We can solve any problem by introducing an extra level of indirection.*

- In particular, we need a trait class and/or type traits
- In the good old days (i.e. 2003) usually defined as one big, fat class
- But now we can do better
  - type traits are *meta-functions* [AG05, 15]

    ```
    struct some_meta_function { typedef ... type; };
    ```
  - at last, functions are free(-standing)

Range Concepts (1)

Single Pass Range (iterator category == Single Pass Iterator)

Meta-functions

```
template< class T >
struct range_value;

template< class T >
struct range_iterator;

template< class T >
struct range_const_iterator;
```

Functions

```
range_iterator<T>::type
begin( T& c );

range_const_iterator<T>::type
begin( const T& c );

range_iterator<T>::type
end( T& c );

range_const_iterator<T>::type
end( const T& c );

bool
empty( const T& c );
```

Range Concepts (2)

Forward Range (iterator category == Forward Iterator)

Meta-functions

```
template< class T >
struct range_difference;

template< class T >
struct range_size;
```

Functions

```
template< class T >
typename range_size<T>::type
size( const T& c );
```

Range Concepts (3)

Bidirectional Range (iterator category == Bidirectional Iterator)
Random Access Range (iterator category == Random Acccess Iter.)

Meta-functions	*Functions*
`template< class T > struct range_reverse_iterator;`	`range_reverse_iterator<T>::type rbegin( T& c );`
`template< class T > struct range_const_reverse_iterator;`	`range_const_reverse_iterator<T>::type rbegin( const T& c );`
	`range_reverse_iterator<T>::type rend( T& c );`
	`range_const_reverse_iterator<T>::type rend( const T& c );`

Algorithms (1)

- Old declaration

```
template< class RandomAccessIterator >
void sort( RandomAccessIterator begin,
           RandomAccessIterator end );
```

- New declaration and definition

```
template< class RandomAccessRange >
void sort( RandomAccessRange& r )
{ sort( boost::begin(r), boost::end(r) ); }
```

- So now we can write `sort( a_vector );` ... but

Algorithms (2)

The Forwarding Problem shows its ugly face again (see eg. [HDA02])

- This fails

```
typedef ... iterator;
pair<iterator,iterator> get_range();
...
sort( get_range() ); // doh!
```

- So we live with the forwarding problem for now
  - it is a *library* problem more than a user problem
  - we add

```
template< class RandomAccessRange >
void sort( const RandomAccessRange& r );
```

Algorithms (3)

So what have our efforts given us?

```
deque<string>                 deq;
pair<iterator,iterator>  get_range();
int                           array[42];
char*                         str = get_c_string();

...
sort( deq );
sort( get_range() );
sort( array );
sort( str );
```

Algorithms (3)

So what have our efforts given us?

```
deque<string>                deq;
pair<iterator,iterator>  get_range();
int                          array[42];
char*                        str = get_c_string();
my_udt                       udt;
...
sort( deq );
sort( get_range() );
sort( array );
sort( str );
sort( udt ); // error, perhaps
```

Supporting User-Defined Types (1)

- If T is a container, it "just works" (eg. `T == boost::array<U,10>`)
- Otherwise we must
  - supply all 5 meta-functions
  - supply 5 basic functions

```
iterator         begin( Range& r );
const_iterator   begin( const Range& r );
iterator         end( Range& r );
const_iterator   end( const Range& r );
size_type        size( const Range& r );
```

  - all other functions are defined on top of these (!)

Supporting User-Defined Types (2)

- ADL customization points are used
- Library code

```
template< class RandomAccessRange >
void sort( RandomAccessRange& r )
{
    using boost::begin;
    using boost::end;
    sort( begin(r), end(r) );
}
```

Supporting User-Defined Types (3)

- Adhere to Interface Principle [Sut00, 133ff]
- Put everything in the right namespace

```
namespace my_namespace
{
    template<>
    struct range_iterator< CString >
    {
        typedef TCHAR* type;
    };
    ...
    TCHAR* begin( CString& r )
    {
        return r.GetBuffer(0);
    }
 }
```

Examples (1)

Functional-style programming [Nie05b]

```
vector<int>                             integers;
...
// old style
tydedef vector<int>::iterator  iterator;

pair<iterator,iterator> p =
    equal_range( integers.begin(), integers.end(), 0 );

for_each( p.first, p.second, some_predicate() );
```

Examples (1)

Functional-style programming [Nie05b]

```
vector<int>                        integers;
...
// old style
tydedef vector<int>::iterator  iterator;

pair<iterator,iterator> p =
    equal_range( integers.begin(), integers.end(), 0 );

for_each( p.first, p.second, some_predicate() );

// new style
for_each( equal_range(integers, 0), some_predicate() );
```

Examples (2)

String functions is a good application area [Dro04]

```
string str1(" hello world! ");
to_upper( str1 );  // str1 == " HELLO WORLD! "
trim( str1 );      // str1 == "HELLO WORLD!"

string str2 =
   to_lower_copy(
      ireplace_first_copy(
         str1, "hello", "goodbye" ));
BOOST_ASSERT( str2 == "goodbye world!" );
```

Examples (3)

"Virtual" Tokenizing is super efficient [Dro04]

```
string str1("hello abc-*-ABC-*-aBc goodbye");

vector< sub_range<string> > find_vector;

ifind_all( find_vector, str1, "abc" );

for( unsigned i = 0u; i != 3; ++i )
    cout « find_vector[i] « " ";

// prints "abc ABC aBc"
```

A trade-off between speed and convenience

Helper Classes (1)

Boost.Range has two helper classes

- `iterator_range<iterator>`
  - used when the most general interface is needed
  - like a better `pair<iterator,iterator>`
  - cannot propagate constness
- `sub_range<Range>`
  - used when support for Single Pass Ranges is unwanted
  - will propagate constness
  - less clumsy to use
- Overloaded functions include «, <, ==, `!=` and `boost::hash_value()`

Helper Classes (2)

Usage examples

- Helper classes "blend in" with normal types

```
string        str    = "hello world";
sub_range<string> r =
    make_iterator_range( begin(str), begin(str) + 5 );

BOOST_CHECK( r == "hello" );
BOOST_CHECK( r != "hell" );
BOOST_CHECK( r < "hello dude" );
BOOST_CHECK( r.front() == 'h' );
BOOST_CHECK( r[0] == 'h' );
```

- Major speed up in word counting (twice as fast(!))

```
typedef boost::sub_range<string>                   match_type;
typedef boost::unordered_map<match_type,int> count_type;
```

Impact on C++ 0x (1)

Expect to see

- All existing algorithms with range interface
- All new algorithms with range interface
- All containers will support

```
cont.assign( range );
cont.insert( cont.begin(), range );
```

- Range adapters that integrate with new for loop

```
std::vector<int*> ptr_vec = ...;
for( int i : ptr_vec | reverse | indirect )
    std::cout « i;
```

- Until then consider (see [Nie05a])

```
BOOST_FOREACH( int i, ptr_vec | reverse | indirect )
    std::cout « i;
```

Impact on C++ 0x (2)

- Template aliases could also be handy
- We can avoid

```
typename range_iterator<Range>::type
```

- Here is how

```
template< class Range >
using iterator =
        typename range_iterator<Range>::type;
template< class Range >
void foo( Range& r )
{ iterator<Range> i = begin(r); ... }
```

- Observation: we still need the underlying meta-functions for
  - the template alias
  - template meta-programming

Summary Part 1

I hope you will remember the following

- Boost.Range is about infrastructure for algorithms
- Algorithmic interfaces can be greatly simplified and we can support functional programming better
- Free-standing traits are superior to monolithic
- Virtual Tokenization can really boost application performance (no pun intended)

Motivation (1)

Why are we here?

- A new student programmer was tested
- He had written a program

```
some_class::~some_class()
{
    map<string,Foo*>::iterator i = m.begin(),
                               e = m.end();
    for( ; i != e; ++i )
        delete i->second; // Hm
}
```

- That was the last straw!

Motivation (2)

If syntactic sugar didn't count, we'd all be programming in assembly language.

- Exception safety
- OO-programming can be painful
  - we need Garbage Collection
- `boost::shared_ptr<T>` will help, but
  - implies syntactic overhead
  - potentially slow
  - what if the objects are not shared?
- Observation: there is no "standard" way to do OO in C++

Motivation (2)

If syntactic sugar didn't count, we'd all be programming in assembly language.

- Exception safety
- OO-programming can be painful
  - we need Garbage Collection
- `boost::shared_ptr<T>` will help, but
  - implies syntactic overhead
  - potentially slow
  - what if the objects are not shared?
- Observation: there is no "standard" way to do OO in C++

I think that object orientedness is almost as much of a hoax as Artificial Intelligence (A. Stepanov)

The Solution

A new set of containers

- One for each existing type, for example
  - `deque<T*>` $\Longrightarrow$ `ptr_deque<T>`
  - `map<Key,T*>` $\Longrightarrow$ `ptr_map<Key,T>`
- Exception-safe implementation
- Hides pointers to minimize problems
- Very efficient implementation
- Containers are "clone-aware"
- A new set of member functions suddenly makes sense in the OO problem domain

Template Interface Comparison

- The good old interface

```
template
<   class T,
    class Allocator = std::allocator<T>
>
class vector;
```

- The shiny new interface

```
template
<   class T,
    class CloneAllocator = heap_clone_allocator,
    class Allocator      = std::allocator<void*>
>
class ptr_vector;
```

Template Interface Comparison

- The good old interface

```
template
<   class T,
    class Allocator = std::allocator<T>
>
class vector;
```

Examples (1)

Pointer Container vs. Container of `smart_ptr<T>`

```
vector< shared_ptr<Foo> >  vec;
ptr_vector<Foo>            ptr_vec;
```

Examples (1)

Pointer Container vs. Container of `smart_ptr<T>`

```
vector< shared_ptr<Foo> >  vec;
ptr_vector<Foo>            ptr_vec;

vec.push_back( shared_ptr<Foo>( new Foo ) );
ptr_vec.push_back( new Foo );
```

Examples (1)

Pointer Container vs. Container of `smart_ptr<T>`

```
vector< shared_ptr<Foo> >  vec;
ptr_vector<Foo>            ptr_vec;

vec.push_back( shared_ptr<Foo>( new Foo ) );
ptr_vec.push_back( new Foo );

vec[0]->foo();
ptr_vec[0].foo();
```

Examples (1)

Pointer Container vs. Container of `smart_ptr<T>`

```
vector< shared_ptr<Foo> >  vec;
ptr_vector<Foo>            ptr_vec;

vec.push_back( shared_ptr<Foo>( new Foo ) );
ptr_vec.push_back( new Foo );

vec[0]->foo();
ptr_vec[0].foo();

vec[0] = shared_ptr<Foo>( new Foo );
ptr_vec.replace( 0, new Foo );
```

Examples (1)

Pointer Container vs. Container of `smart_ptr<T>`

```
vector< shared_ptr<Foo> >  vec;
ptr_vector<Foo>            ptr_vec;

vec.push_back( shared_ptr<Foo>( new Foo ) );
ptr_vec.push_back( new Foo );

vec[0]->foo();
ptr_vec[0].foo();

vec[0] = shared_ptr<Foo>( new Foo );
ptr_vec.replace( 0, new Foo );

(*vec.begin())->foo();
ptr_vec.begin()->foo();
```

Examples (2)

Pointer Container vs. Container of `smart_ptr<T>` (cont.)

```
vector< shared_ptr<Foo> >  vec2 = vec; // shallow cpy
ptr_vector<Foo>            ptr_vec2 = ptr_vec; // error
// instead: deep copy
ptr_vector<Foo>        ptr_vec2 = ptr_vec.clone();
// instead: transfer ownership
ptr_vector<Foo>        ptr_vec2 = ptr_vec.release();
```

Examples (2)

Pointer Container vs. Container of `smart_ptr<T>` (cont.)

```
vector< shared_ptr<Foo> >  vec2 = vec; // shallow cpy
ptr_vector<Foo>         ptr_vec2 = ptr_vec; // error
// instead: deep copy
ptr_vector<Foo>     ptr_vec2 = ptr_vec.clone();
// instead: transfer ownership
ptr_vector<Foo>     ptr_vec2 = ptr_vec.release();

shared_ptr<Foo> foo = vec.back();
vec.pop_back();
ptr_vector<Foo>::auto_type foo = ptr_vec.pop_back();
```

We shall see more differences later ...

Exception Safety (1)

Three types of guarantee [Sut00, Sut02]

- Basic guarantee (invariants are preserved)

```
vector<Foo> vec, vec2;
...
vec.insert( vec.end(), vec2.begin(), vec2.end() )
```

- Strong guarantee (roll-back guarantee)

```
vec.push_back( Foo() );
```

- Nothrow guarantee

```
cout « vec[0];
```

Exception Safety (2)

- All containers are implemented as wrappers around standard containers
  - `ptr_vector<T>` uses `vector<void*>` internally etc.
  - destructor deletes objects
- Mental picture

```
template< class T >
class ptr_vector
{
    std::vector<void*> vec_;
public:
    ~ptr_vector(); // delete objects
    // ... much more
   typedef <something> iterator;
};
```

Exception Safety (3)

- Consider how to implement

```
ptr_vector<T>::push_back( T* );
```

- Take 1

```
void push_back( T* r )
{
    vec_.push_back( r );
}
```

- Since `vector<void*>::push_back()` has the strong guarantee, then so must this, right?
- Why is this implementation naive?

Exception Safety (4)

- Recall that `vector<T*>::push_back()` can throw
- Take 2

```
void push_back( T* r )
{
    auto_ptr<T> ptr( r );
    vec_.push_back( r );
    ptr.release();
}
```

- Voila, now we fulfill the strong guarantee
- Exception safety here is relatively costly
  - however, dwarfed by cost of heap-allocation

Exception Safety (5)

Let us try something harder

```
template< class Iter >
void insert( iterator before, Iter first, Iter last )
{
      while( first != last )
      {
          vec_.insert( before, new T( *first ) );
          ++first;
      }
}
```

This implementation has three errors and one flaw (!)—can you spot them?

Exception Safety (5)

Let us try something harder

```
template< class Iter >
void insert( iterator before, Iter first, Iter last )
{
      while( first != last )
      {
          vec_.insert( before, new T( *first ) );
          ++first;
      }
}
```

This implementation has three errors and one flaw (!)—can you spot them?

Exception Safety (5)

Let us try something harder

```
template< class Iter >
void insert( iterator before, Iter first, Iter last )
{
      while( first != last )
      {
            vec_.insert( before, new T( *first ) );
            ++first;
      }
}
```

This implementation has three errors and one flaw (!)—can you spot them?

Exception Safety (5)

Let us try something harder

```
template< class Iter >
void insert( iterator before, Iter first, Iter last )
{
      while( first != last )
      {
          vec_.insert( before, new T( *first ) );
          ++first;
      }
}
```

This implementation has three errors and one flaw (!)—can you spot them?

Exception Safety (5)

Let us try something harder

```
template< class Iter >
void insert( iterator before, Iter first, Iter last )
{
      while( first != last )
      {
          vec_.insert( before, new T( *first ) );
          ++first;
      }
}
```

This implementation has three errors and one flaw (!)—can you spot them?

- Flaw: we do exploit size of range can be found

Exception Safety (6)

Take 2

```
template< class Iter >
void insert( iterator before, Iter first, Iter last )
{
    while( first != last )
    {
        auto_ptr<T> ptr( new T( *first ) );
        before = vec_.insert( before, ptr.get() );
        ++before; // to preserve order
        ++first;
        ptr.release();
    }
}
```

We have got the basic guarantee, but the flaw is still there

Exception Safety (7)

Problems

- Why don't we just call

```
vec_.reserve(distance(first,last) + vec_.size());
```

  ?

- Several reasons
  - `Iter` could be an Input Iterator (we'll ignore this)
  - `reserve()` can invalidate `before`
  - `reserve()` is specific to `vector`
- Where do we go from here?
  - traits classes
  - "smart" iterators
  - generic solution

Exception Safety (8)

The generic alternative has a superior design

- Requires a small helper class
- But implementation is fairly simple

```
template< class Iter >
void insert( iterator before, Iter first, Iter last )
{
    size_t n = distance(first, last);
    scoped_deleter<T> sd( n );

    for( ; first != last; ++first )
        sd.add( new T( *first ) );

    vec_.insert( before, sd.begin(), sd.end() );
    sd.release();
}
```

Exception Safety (9)

- Now implementation works with
  - `list<T*>`
  - `deque<T*>`
  - `vector<T*>`
  - standard compliant sequences (eg. `slist<T*>`)
- We've got the strong guarantee
  - the overhead is one heap-allocation (no problem)
- Implementing `scoped_deleter<T>` consists of
  - allocate/deallocate buffer in constructor/destructor
  - keep track of current size in `add()`
  - delete objects in destructor
  - add `begin()/end()` functions

The Clonable Concept (1)

- But one line remains problematic

```
sd.add( new T( *first ) ) );
```

- Clearly a customization point since
  - not all types are allocated with default `operator new()`
  - not all types can be copy-constructed
  - copy-construction of *polymorphic* types is bad design (cf. explicit class proposal [GG04])
- Enter Clonable concept

```
template< class T >
inline T* new_clone( const T& r )
{ return new T( r ); }
template< class T >
inline void delete_clone( const T* r )
{ checked_delete( r ); }
```

The Clonable Concept (2)

- Recall interface

```
template
<   class T,
    class CloneAllocator = heap_clone_allocator,
    class Allocator      = std::allocator<void*>
>
class ptr_vector;
```

- Yet another concept: Clone Allocator
    - allows different types of cloning (shallow, deep)
    - creates ADL customization point
    - stateless (unlike `allocator<T*>`)
    - orthogonal to Allocator concept

The Clonable Concept (3)

- The default clone allocator

```
struct heap_clone_allocator
{
    template< class U >
    static U* allocate_clone( const U& r )
    {
        return new_clone( r ); // ADL hook
    }
    template< class U >
    static void deallocate_clone( const U* r )
    {
        delete_clone( r ); // ADL hook
    }
};
```

- Empty functions can be easily optimized away

New Stuff (1)

Iterators have changed

- Wraps underlying `void*` iterators and apply cast and indirection

```
T& r = *i;  // T& r  = **i;
i->foo();   // (*i)->foo();
```

- New map-iterators

```
ptr_map<string,T>::iterator i = m.begin(),
                            e = m.end();
for( ; i != e; ++i )
{
    i->foo(); // call T::foo()
    if( i.key() == "bar" )
        ...
}
```

- The goal is to hide the pointers

New Stuff (2)

Handling nulls

- By default containers will throw if 0 is added
- Consider if Null Object Pattern can be used [Hen02]
  - no special cases
  - could be faster
- It is not always applicable, so then what?

```
ptr_list< nullable<T> > list;
list.push_back( 0 ); // ok
...
for( ; i != e; ++i )
{
    if( !is_null(i) )
        i->foo();
}
```

New Stuff (2)

Summary of new member functions

```
auto_type   release( iterator where );
auto_type   replace( iterator where, T* );
auto_type   replace( size_type at, T* );

auto_ptr<ptr_container> release();
auto_ptr<ptr_container> clone() const;

void transfer( iterator before, iterator first,
               iterator last, ptr_container& from );
void transfer( iterator before, ptr_container& from );
```

Copy semantics are removed, fewer constructors, no `resize()`

Pointer Containers vs. Containers of Smart Pointers(1)

The `unique_ptr<T>` smart pointer

- Is being proposes as a replacement for `auto_ptr<T>` in C++0x [ADG+05]
- Can be placed in containers because of language support for move-semantics
- Hence `vector< unique_ptr<T> >` is a direct competitor to `ptr_vector<T>`
- There are many differences though, for example

```
// vector< unique_ptr<T> >
other.push_back( move( vec.back() ) );
vec.pop_back();

// ptr_vector<T>
other.push_back( vec.pop_back().release() );
```

Pointer Containers vs. Containers of Smart Pointers(2)

Comparison

- `vector< unique_ptr<T> >`
  - is considerably easier to implement (nothing to do)
  - has a well-known interface
- `ptr_vector<T>`
  - has an indirected interface
  - can guarantee only `void*` instantiations
  - supports many new functions for the new domain
  - integrates support for Clonable types
  - supports containers without nulls
  - has stronger exception safety guarantees
  - has a "cleaner" syntax
- Conclusion: Pointer Containers are here to stay

Examples—Genetic Programming (1)

A class hierarchy

```
class royal_person : noncopyable
{
    virtual string        do_speak() const = 0;
    virtual royal_person* do_clone() const = 0;
public:
    virtual ~royal_person() { }

    string        speak() const { return do_speak(); }
    royal_person* clone() const { return do_clone(); }
};
```

Clonability

```
inline royal_person* new_clone( const royal_person& r )
{ return r.clone(); }
```

Examples—Genetic Programming (2)

Time for human(e) export

```
typedef ptr_vector<royal_person> royal_family;
royal_family danish_family, french_family,
             american_family;
danish_family.push_back( new queen("Magrethe") );
danish_family.push_back(
  new count("Friedrich Richard Oscar Jefferson"
            " von Pfeil und Klein-Ellguth") );
```

Examples—Genetic Programming (2)

Time for human(e) export

```
typedef ptr_vector<royal_person> royal_family;
royal_family danish_family, french_family,
             american_family;
danish_family.push_back( new queen("Magrethe") );
danish_family.push_back(
  new count("Friedrich Richard Oscar Jefferson"
            " von Pfeil und Klein-Ellguth") );
// God forbid
french_family = danish_family.clone();
```

Examples—Genetic Programming (2)

Time for human(e) export

```
typedef ptr_vector<royal_person> royal_family;
royal_family danish_family, french_family,
             american_family;
danish_family.push_back( new queen("Magrethe") );
danish_family.push_back(
  new count("Friedrich Richard Oscar Jefferson"
            " von Pfeil und Klein-Ellguth") );
// God forbid
french_family = danish_family.clone();

// Wishful thinking
american_family = danish_family.release();
```

Examples—Genetic Programming (2)

Time for human(e) export

```
typedef ptr_vector<royal_person> royal_family;
royal_family danish_family, french_family,
             american_family;
danish_family.push_back( new queen("Magrethe") );
danish_family.push_back(
  new count("Friedrich Richard Oscar Jefferson"
            " von Pfeil und Klein-Ellguth") );
// God forbid
french_family = danish_family.clone();

// Wishful thinking
american_family = danish_family.release();

ptr_vector<royal_family> royals;
royals.push_back( french_family.release() );
royals.push_back( american_family.release() );
royals.clear();
```

Examples—A Tree Container (1)

An minimal overhead tree

```
template< class Node, size_t N >
class n_ary_tree : noncopyable
{
    typedef n_ary_tree<Node,N>        this_type;
    typedef ptr_array<this_type,N>    tree_t;

    tree_t              tree;
    Node                data;
public:
    void set_data( const Node& r );
    void print( ostream&, string indent = "  " );

    template< size_t idx >
    void set_child( this_type* r )
    { tree. template replace<idx>(r); }
};
```

Examples—A Tree Container (2)

A straightforward recursion

```
void n_ary_tree<Node,N>::print( ostream& out,
                                string indent )
{   out « indent « data « "\n";
    indent += "  ";
    for( size_t i = 0; i != N; ++i )
        if( !tree.is_null(i) )
            tree[i].print( out, indent  + "  " ); }
```

We have compile-time bounds check

```
typedef n_ary_tree<std::string,2> binary_tree;
binary_tree tree;
tree.set_data( "root" );
tree.set_child<0>( new binary_tree( "left" ) );
tree.set_child<1>( new binary_tree( "right" ) );
tree.print( std::cout );
```

Summary Part 2

I hope the following will be remembered

- Boost.Pointer Container is about making OO programming easy
- The three guarantees of exception safety (basic,strong,nothrow)
- Good libraries imply users rarely have to worry about exception safety
- The advantages to OO domain specific idioms
    - `boost::noncopyable`
    - private virtual functions [Sut01]
    - the Clonable concept
- When it is appropriate to use Boost.Pointer Container

By the way

- I’m also one of the proposers of Contract Programming (a.k.a. Design by Contract) for C++0x.
- Please ask questions if you’re interested (!).
- I could give a "private" presentation if people wanted it.

David Abrahams, Peter Dimov, Doug Gregor, Howard E. Hinnant, Andreas Hommel, and Alisdair Meredith.
Impact of the rvalue reference on the Standard Library.
http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2005/n1771.html, 2005.

David Abrahams and Aleksey Gurtovoy.
C++ Template Metaprogramming.
Addison-Wesley, 2005.

Pavol Droba.
Boost String Algorithms Library.
http://www.boost.org/doc/html/string_algo.html, 2004.

Francis Glassborow and Lois Goldthwaite.
explicit class and default definitions.

http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2004/n1702.pdf, 2004.

Howard E. Hinnant, Peter Dimov, and Dave Abrahams.

A Proposal to Add Move Semantics Support to the C++ Language .

http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2002/n1377.htm, 2002.

Kevlin Henney.

Null Object.

http://www.two-sdg.demon.co.uk/curbralan/papers/europlop/NullObject.pdf, 2002.

Eric Niebler.

Boost Foreach Proposal.

http://boost-sandbox.sourceforge.net/vault, 2005.

Eric Niebler.
Range Extension 1.0 Documentation.
http://boost-sandbox.sf.net/libs/range_ex, 2005.

Herb Sutter.
Exceptional C++.
Addison-Wesley, 2000.

Herb Sutter.
Virtuality.
http://www.gotw.ca/publications/mill18.htm, 2001.

Herb Sutter.
More Exceptional C++.
Addison-Wesley, 2002.