Functional Programming with Boost

Pure Functional Programming

- A paradigm for computing with functions
- Based on Lambda Calculus
- No data mutation
  - ✓ syntactically uniform
  - ✓ mathematical
  - ✓ easy to reason about
- C++ is not “pure”

STL and the Functional Paradigm

- Remember this?

```
// For each int i in s, print i * 3:
std::transform(
    s.begin(), s.end(), std::ostream_iterator<int>(std::cout, " "),
    std::bind2nd(std::multiplies<int>(), 3) );
```

Pros	Cons
✓Efficient	✗Hard to write
✓Idiomatic	✗Hard to read

Lambda Functions

- Using Boost.Lambda

```
// For each int i in s, print i * 3:
std::for_each(
    s.begin(), s.end(), std::cout << ( _1 * 3 ) << " " );
```

Boost.Lambda helps STL deliver on the promise of Functional Programming

Demystifying Lambda

- Consider a simple predicate object ...

```
template<class T>
struct less_than
{
    less_than(T const & x) : right(x) {}

    bool operator()(T const & left) const
    {
        return left < this->right;
    }

    T right;
};
```

- ... used as follows:

```
std::find_if(v.begin(), v.end(), less_than<int>(2));
```

Demystifying Lambda

- Now create a placeholder type

```
struct lambda_variable {};
lambda_variable const _x_;

template<class T>
less_than<T> operator<(lambda_variable, T const & t)
{
    return less_than<T>(t);
}
```

- Now we can write lambda expressions!

```
std::find_if(v.begin(), v.end(), _x_ < 2);
```

Use Phoenix Instead of Lambda

- This is the future of Boost.Lambda
- See http://boost.org/libs/spirit/phoenix
- Compatible syntax, but works better
- Preparation:

```
#include <boost/spirit/include/phoenix.hpp>
using namespace boost::phoenix::arg_names;
```

Binding Member Functions

Resize all strings in a vector<string>:

- with STL binders:

```
std::for_each( s.begin(), s.end(),
    std::bind2nd( std::mem_fun_ref( &std::string::resize ),
new_len ) );
```

- with Boost.Bind:

```
std::for_each( s.begin(), s.end(),
 std::tr1::bind( &std::string::resize, _1, new_len ) );
```

Binding Data Members, Too!

Copy all values from a map<int,std::string>:

- with STL binders:

```
?
```

- with Boost.Bind:

```
std::transform( m.begin(), m.end(), output,
    boost::bind( &std::pair<int const,std::string>::second, _1 ) );
```

Example

- A loan can tell you its risk over a period:

```
class loan
{
public:
    virtual float risk(int years) = 0;
};

std::vector<loan *> loans;
```

- Total risk calculation:

```
float total = 0.0f;
for (int i = 0; i < loans.size(); ++i)
    total += loans[i]->risk(years);
```

Nested Binds

- Total risk calculation:

```
float total = 0.0f;
for (int i = 0; i < loans.size(); ++i)
    total += loans[i]->risk(years);
```

- With Boost.Bind:

```
total = std::accumulate( loans.begin(), loans.end(), 0.0f,
    bind( std::plus<float>(), _1, bind( &loan::risk, _2, years ) ) );
```

- With Boost.Phoenix:

```
using namespace boost::phoenix::arg_names;
total = std::accumulate( loans.begin(), loans.end(), 0.0f,
    _1 + bind( &loan::risk, _2, years ) ) );
```

Return Type Deduction

- Problem: What should the return type be?

```
template< class T, class U >
??????? add_stuff( T const & t, U const & u )
{
    return t + u;
}

add_stuff( short(1), int(2) );               // int(3)
add_stuff( std::string("hell"), 'o' )        // std::string("hello")
```

Return type deduction is hard!
Boost.Bind doesn't try to solve it.

Boost.Bind and function objects

- Return type deduction is hard, so function objects have to help

```
boost::bind(std::plus<int>(), _1, 42)
```

std::plus is Adaptable

```
template <class T>
struct plus
{
    typedef T result_type;

    T operator()(T const& x, T const& y) const
    {
        return x + y;
    }
};
```

Bind and Relational Operators

- Find a pair for which p.second == 42

```
std::vector< std::pair<int, int> > v;

std::find_if( v.begin(), v.end(),
    42 == std::tr1::bind( &std::pair<int, int>::second, _1 ) );
```

What is special about the *relational operators* that lets Boost.Bind horn in on Lambda/Phoenix territory like this?

C++0x Alert

- This will get easier!

```
template <class T>
T make();

template< class T, class U >
decltype(make<T>() + make<U>())
add_stuff( T const & t, U const & u )
{
    return t + u;
}
```

C++0x Alert

- This will get <u>much</u> easier!

```
template< class T, class U >
auto add_stuff( T const & t, U const & u ) -> decltype( t + u )
{
    return t + u;
}
```

Binding References

- Normally bind stores everything by value:

```
bind( std::plus<int>(), 42, _1 ); // stores a copy of 42
```

- ...which can get expensive:

```
void f(huge_matrix const& x)
{
    bind(std::plus<huge_matrix>(), x, _1); // stores a copy of x
}
```

- ...and blocks mutation:

```
int x = 0;
std::for_each( s.begin(), s.end(), bind(add_if_negative, x, _1) );
// x unmodified here.
```

Binding References

- Use cref() to pass by const reference

```
bind( std::plus<int>(), cref(42), _1 ); // sorta pointless
```

- (which saves copies):

```
void f(huge_matrix const& x)
{
    bind(std::plus<huge_matrix>(), cref(x), _1); // huge win!
}
```

- ...and ref() to allow mutation:

```
int x = std::numeric_limits<int>::max();
std::for_each( s.begin(), s.end(), bind(add_if_negative, ref(x), _1) );
// x contains sum of negative element in s
```

Exercise: Use Boost/TR1 Bind

```
class image;

class animation
{
public:

    void advance(int ms);
    bool inactive() const;
    void render(image & target) const;
};

std::vector<animation> anims;

template<class C, class P>
void erase_if(C & c, P pred)
{
    c.erase(
            std::remove_if(c.begin(), c.end(), pred),
            c.end()
            );
}
```

```
void update(int ms)
{
    // use std::for_each and bind to advance
    // each animation by ms
    std::for_each(anims.begin(), anims.end(),
                            … );

    // erase all inactive animations using erase_if
    erase_if( … );
}

void render(image & target)
{
    // render each animation against target
}
```

Exercise

(in progress)

- use std::for_each and bind to advance each animation by ms
- erase all inactive animations using erase_if
- render each animation against target

Storing Lambdas and Binds

- Problem: how to save a bind expression?

```
unspecified-callable-type add42 = bind( std::plus<int>(), 42, _1 );
cout << add42( x ) << endl;  // prints x + 42
```

```
boost::_bi::bind_t<
    boost::_bi::unspecified
  , std::plus<int>
  , boost::_bi::list2<boost::_bi::value<int>
   , boost::arg<1> > >
```

Storing Lambdas and Binds

- Problem: how to save a bind expression?

```
boost::function<int(int)> add42 = bind( std::plus<int>(), 42, _1 );
cout << add42( x ) << endl;  // prints x + 42
```

- Boost.Function saves Lambdas, too:

```
add42 = 42 + _1;
```

Boost.Function

- Usage: `boost::function<R ( A1, A2, … An )>`
- Holds/forwards to anything callable that
  - accepts arguments A1, A2, … A*n* and
  - whose result can be converted to R
- For example:

```
typedef boost::function<int (int, int)> func;
func f0( &min<int> );                // function pointer
func f1( std::less<long>() );        // function object
func f2( std::less<int>() );
func f3 = f0;     int x = f3(5, 6);  // x == 5
f3 = f2;          int y = f3(5, 6);  // x == 1
std::vector<func> v;
```

Demystifying Boost.Function

```
typedef int R,A0,A1;

struct wrapper_base
{
    virtual R invoke(A0,A1) = 0;
    virtual wrapper_base* clone() const = 0;
    virtual ~wrapper_base() {}
};

template <class F>
struct wrapper<F> : wrapper_base
{
    wrapper(F f) : f(f) {}

    R invoke(A0 a0, A1 a1) { return f(a0,a1); }

    wrapper_base* clone() const
    { return new wrapper(f); }

    F f;
};
`
```

```
struct function
{
    template <class F>
    function(F f)
      : impl( new wrapper<F>(f) )  {}

    R operator()(A0 a0, A1 a1) const
    { return impl->invoke(a0,a1); }

    function(function const& rhs)
      : impl( rhs.impl->clone() )  {};

    function& operator=(function rhs)
    { std::swap(impl, rhs.impl); return *this; }

private:
    std::auto_ptr<
        wrapper_base
    > impl;
};
```

Demystifying Boost.Function

```
template <class R, class A0, class A1>
struct wrapper_base
{
      virtual R invoke(A0,A1) = 0;
      virtual wrapper_base* clone() const = 0;
      virtual ~wrapper_base() {}
};

template <class F, class R, class A0, class A1>
struct wrapper<F> : wrapper_base<R,A0,A1>
{
      wrapper(F f) : f(f) {}

      R invoke(A0 a0, A1 a1) { return f(a0,a1); }

      wrapper_base* clone() const
      { return new wrapper(f); }

      F f;
};
```

```
template <class R, class A0, class A1>
struct function<R(A0,A1)>
{
      template <class F>
      function(F f)
        : impl( new wrapper<F,R,A0,A1>(f) )  {}

      R operator()(A0 a0, A1 a1) const
      { return impl->invoke(a0,a1); }

      function(function const& rhs)
        : impl( rhs.impl->clone() )  {};

      function& operator=(function rhs)
      { std::swap(impl, rhs.impl); return *this; }

private:
      std::auto_ptr<
           wrapper_base<R,A0,A1>
      > impl;
};
```

Demystifying Boost.Function

```
template <class R,
struct wrapper_bas
{
    virtual R invoke(A0,A1) = 0;
    virtual wrapper_base* clone() const = 0;
    virtual ~wrapper_base() {}
};

template <class F, class R, class A0, class A1>
struct wrapper<F> : wrapper_base<R,A0,A1>
{
    wrapper(F f) : f(f) {

    R invoke(A0 a0, A1 a1)                }

    wrapper_base* clone() const
    { return new wrapper(f); }

    F f;
};
```

```
template <class R, class A0, class A1>
struct function<R(A0,A1)>
{
    template <class F>
    function(F f)
      : impl( new wrapper<F,R,A0,A1>(f) )  {}

    R operator()(A0 a0, A1 a1) const
    { return impl->invoke(a0,a1); }

    function(function const& rhs)
      : impl( rhs.impl->clone() )  {};

    function& operator=(function rhs)
    { std::swap(impl, rhs.impl); return *this; }

private:
    std::auto_ptr<
        wrapper_base<R,A0,A1>
    > impl;
};
```

“type erasure” happens here…

…and here

A Version Without Virtual Functions

```
template <class T>
void destroy(void const* p)
{  delete static_cast<T const*>(p);  }

template <class T>
void* clone(void const* p)
{ return new T(  *static_cast<T const*>(p)  );  }

template <class F, class R, class A0, class A1>
R invoke(void* f, A0 a0, A1 a1)
{ return (  *static_cast<F*>(f)  )(a0,a1);  }

template <class R, class A0, class A1>
struct function<R(A0,A1)>
{
      template <class F>
      function(F f)
        : impl( new F(f) ), invoker(invoke<F,R,A0,A1>)
        , destructor(destroy<F>), cloner(clone<F>)
      {}

      R operator()(A0 a0, A1 a1) const
      { return invoker(impl, a0,a1); }
      …
```

```
      function(function const& rhs)
        : impl( rhs.cloner(rhs.impl) ),invoker(rhs.invoker)
        , destructor(rhs.destructor), cloner(rhs.cloner)
      {}

      function& operator=(function const& rhs)
      {
            void* r = rhs.cloner(rhs.impl);
            destructor(impl);
            impl = r;
            invoker = rhs.invoker;
            destructor = rhs.destructor;
            cloner = rhs.cloner;
            return *this;
      }

      ~function() { destructor(impl); }

private:
      void* impl;
      R (*invoker)(void*,A0,A1);
      void (*destructor)(void const*);
      void* (*cloner)(void const*);
};
```

Results

- Tested 100,000,000 invocations

Code Size	Virtual Functions	Function Pointers
MSVC-7.1	11725	5194
gcc-4.0.0	6397	3141

Run Time	Virtual Functions	Function Pointers
MSVC-7.1	1.66s	1.73s
gcc-4.0.0	2.38s	2.23s

C++0x Alert:

- This will get much easier!

```
auto add42  = bind( std::plus<int>(), 42, _1 );
auto ladd42 = 42 + boost::lambda::_1;
```

C++0x Alert:

- ...and more efficient!

```
auto add42  = bind( std::plus<int>(), 42, _1 );
auto ladd42 = 42 + boost::lambda::_1;
```

```
boost::_bi::bind_t<
    boost::_bi::unspecified
  , std::plus<int>
  , boost::_bi::list2<boost::_bi::value<int>
  , boost::arg<1> > >
```

C++0x Alert:

- ...and more efficient!

```
auto add42  = bind( std::plus<int>(), 42, _1 );
auto ladd42 = 42 + boost::lambda::_1;
```

```
boost::_bi::bind_t<
    boost::_bi::unspecified
  , std::plus<int>
  , boost::_bi::list2<boost::_bi::value<int>
  , boost::arg<1> > >
```

Other uses for Boost.Function

- Decoupling components
  - powerful and stable APIs
- Generic callbacks, eg.:
  - Boost.Threads
  - Boost.Signals

Ad hoc polymorphism

The benefits of OOP without the goop

Member Access

- bind() syntax isn't always ideal

```
int sum_second( vector< pair<int,int> * > const & v ) {
    return std::accumulate( v.begin(), v.end(), 0,
        _1 + bind( &pair<int,int>::second, _2 ) );
}
```

- Wouldn't this be better?

```
int sum_second( vector< pair<int,int> * > const & v ) {
    return std::accumulate( v.begin(), v.end(), 0,
        _1 + _2 ->* &pair<int,int>::second );  // OK!
}
```

- (not ideal either, of course)

A Lambda Gotcha!

- What's wrong with this?

```
void display_ints( vector< int > const & v ) {
    std::for_each( v.begin(), v.end(),
        std::cout << '+' << _1 );
}
```

Displays: "+1234"

- Deferring evaluation

```
void display_ints( vector< int > const & v ) {
    std::for_each( v.begin(), v.end(),
        std::cout << val('+') << _1 );
}
```

Displays: "+1+2+3+4"

Example: Count the Elements

- Closure
  - A function bound to a lexical scope
  - C++ doesn't have them ☹
- Achieving closure with Boost.Phoenix

```
template< class It >
int count_range( It begin, It end ) {
    int i = 0;
    std::for_each( begin, end, i = i + 1 );
    return i;
}
```

Error! Not a lambda!

Example: Count the Elements

- Closure
  - A function bound to a lexical scope
  - C++ doesn't have them ☹
- Achieving closure with Boost.Phoenix

```
template< class It >
int count_range( It begin, It end ) {
    int i = 0;
    std::for_each( begin, end, i = i + val(1) );
    return i;
}
```

Error! Can't assign lambda to int!

Example: Count the Elements

- Closure
  - A function bound to a lexical scope
  - C++ doesn’t have them ☹
- Achieving closure with Boost.Phoenix

```
template< class It >
int count_range( It begin, It end ) {
    int i = 0;
    std::for_each( begin, end, ref(i) = i + val(1) );
    return i;
}
```

Wrong! Evaluates as "i = 0 + 1"

Example: Count the Elements

- Closure
  - A function bound to a lexical scope
  - C++ doesn't have them ☹
- Achieving closure with Boost.Phoenix

```
template< class It >
int count_range( It begin, It end ) {
    int i = 0;
    std::for_each( begin, end, ref(i) = ref(i) + 1 );
    return i;
}
```

OK!

Example: Count the Elements

- Closure
  - A function bound to a lexical scope
  - C++ doesn't have them ☹
- Achieving closure with Boost.Phoenix

```
template< class It >
int count_range( It begin, It end ) {
    int i = 0;
    std::for_each( begin, end, ref(i) = cref(i) + 1 );
    return i;
}
```

OK!

Example: Count the Elements

- Closure
  - A function bound to a lexical scope
  - C++ doesn't have them ☹
- Achieving closure with Boost.Phoenix

```
template< class It >
int count_range( It begin, It end ) {
    int i = 0;
    std::for_each( begin, end, ++ref(i) );
    return i;
}
```

Also OK!

Example: Count the Elements

- Closure
  - A function bound to a lexical scope
  - C++ doesn't have them ☹
- Achieving closure with Boost.Phoenix

```
template< class It >
int count_range( It begin, It end ) {
    int i = 0;
    std::for_each( begin, end, i += val(1) );
    return i;
}
```

Also OK!

Advanced Lambdas

- A Phoenix for loop:

```
int a[5][10] = {};
int i;

std::for_each( a, a+5,
    for_( ref(i) = 0, cref(i) < 10, ++ref(i) )
    [
        _1[ cref(i) ] += 1
    ]
);
```

Condition

Increment

Initialization

Loop body

Advanced Lambdas

- Some other Boost.Phoenix control structures:
  - if_(condition)[then_part]
  - if_(condition)[then_part].else_[else_part]
  - while_(condition)[body]
  - do_[body].while_(condition)
  - and, uh…

Advanced Lambdas

- Yes, exceptions:

```
for_each(
  a.begin(), a.end(),
  try_[
    bind(foo, _1),                 // foo may throw
  ]
  .catch_<foo_exception>() [
      cout << val("Caught foo_exception: ")
           << "foo was called with argument = "   < _1
  ]
  .catch_<std::exception>( _e ) [
      cout << val("Caught std::exception: ")
           << bind(&std::exception::what, _e),
      throw_( construct<bar_exception>(_1) )
  ]
  .catch_all  [
      cout << val("Unknown"),
      throw_()
  ]
);
```

- Actually, not yet with Phoenix
- Boost.Lambda can do it
- Use Phoenix anyway ☺

Lambda/Phoenix Gotcha: Rvalues

- Unnamed temporaries not allowed as actual arguments

```
(std::cout << _1) (3);                    // ERROR!
```

- Boost.Lambda has make_const:

```
(std::cout << _1) (make_const(3));        // OK!
```

- You can write make_const yourself for Phoenix

```
template <class T> T const&
make_const(T const& x)
{ return x; }
```

- …if you need it. Soon you won’t, for 1st 3 args boostpro computing

Exercise: Using Phoenix

- Write one-line expressions to
  - Initialize a vector of integers with the numbers 0 … 99
  - Square the elements of a vector
  - Calculate the sum of squares of a vector
  - Print all even numbers in a vector of integers
- Use only STL algorithms and function objects with Boost.Phoenix
- Do not
  - Use any explicitly written function objects
  - Write any explicit loop
- Bonus: How readable can you make the code?