Posted November 22, 2010 by [Eric Niebler](http://ericniebler.com), under [Boost](http://cpp-next.com/archive/category/boost/), [Functional Programming](http://cpp-next.com/archive/category/functional-programming/)

**Expressive C++: Trouble With Tuples**

This entry is part of a series, [Expressive C++»](javascript:;) Entries in this series:

1. [Expressive C++: Introduction](http://cpp-next.com/archive/2010/08/expressive-c-introduction/)
2. [Expressive C++: Playing with Syntax](http://cpp-next.com/archive/2010/09/expressive-c-playing-with-syntax/)
3. [Expressive C++: Why Template Errors Suck and What You Can Do About It](http://cpp-next.com/archive/2010/09/expressive-c-why-template-errors-suck-and-what-you-can-do-about-it/)
4. [Expressive C++: A Lambda Library in 30 Lines (Part 1 of 2)](http://cpp-next.com/archive/2010/10/expressive-c-expression-extension-part-one/)
5. [Expressive C++: A Lambda Library in 30 Lines (Part 2 of 2)](http://cpp-next.com/archive/2010/10/expressive-c-expression-extension-part-two/)
6. [Expressive C++: Fun With Function Composition](http://cpp-next.com/archive/2010/11/expressive-c-fun-with-function-composition/)
7. Expressive C++: Trouble With Tuples
8. [Expressive C++: Expression Optimization](http://cpp-next.com/archive/2011/01/expressive-c-expression-optimization/)

Welcome to the latest in a series of articles about Embedded Domain-Specific Languages and Boost.Proto, a library for implementing them in C++. Brace yourself; this article is going to range all over, from Haskell-style pattern matching to [TR1](http://en.wikipedia.org/wiki/C%2B%2B_Technical_Report_1)‘s tuples, and draw from the function composition technique from the [last article](http://cpp-next.com/archive/2010/11/expressive-c-fun-with-function-composition/)—all to enhance the 30-line lambda EDSL we’ve been developing—which, when the dust settles, will be far more than 30 lines. By the end, we will know how to compose non-trivial computations on expression trees from simple building blocks, claim more functional programming territory for our own, and even have a slightly more useful lambda library to show for our efforts.

This article got longer than I would have liked. Put on a pot of coffee and get comfortable. This might take a while.

**The Tiny Lambda Library**

A few articles back, we developed a little lambda library: a library for creating anonymous functions in-place. (The original articles are [here](http://cpp-next.com/archive/2010/10/expressive-c-expression-extension-part-one/) and [here](http://cpp-next.com/archive/2010/10/expressive-c-expression-extension-part-two/).) Since I’ll be referring back to it, here’s the solution so far:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | #include <algorithm>  #include <boost/proto/proto.hpp>  using namespace boost;    // A type recognized by the Lambda algorithm below  // as a placeholder for a user-supplied argument.  struct arg1\_tag {};  typedef proto::terminal<arg1\_tag> arg1\_term;    // Forward-declare our custom expression wrapper.  template<typename ProtoExpr> struct lambda\_expr;    // Define a lambda domain and its generator, which  // wraps all new expressions our custom wrapper.  struct lambda\_domain :  proto::domain< proto::pod\_generator<lambda\_expr> >  {};    // A Proto algorithm for evaluating lambda  // expressions.  struct Lambda :  proto::or\_<  // When evaluating the arg1 terminal,  // return the state.  proto::when<arg1\_term, proto::\_state>,  // Otherwise, do the "default" thing.  proto::otherwise< proto::\_default< Lambda > >  >  {}; |
| 30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58 | // A lambda is an expression with an operator()  // that evaluates the lambda.  template<typename ProtoExpr>  struct lambda\_expr  {  BOOST\_PROTO\_BASIC\_EXTENDS(  ProtoExpr, lambda\_expr, lambda\_domain)  BOOST\_PROTO\_EXTENDS\_ASSIGN()  BOOST\_PROTO\_EXTENDS\_SUBSCRIPT()    // So that boost::result\_of can calculate  // the return type of this lambda expression.  template<typename Sig> struct result;    template<typename T, typename A1>  struct result<T(A1)> :  boost::result\_of<Lambda(T const&, A1 const&)>  {};    // Evaluate the lambda expressions  template<typename A1>  typename result<lambda\_expr(A1)>::type  operator()(A1 const & a1) const  {  return Lambda()(\*this, a1);  }  };    lambda\_expr<arg1\_term::type> const arg1 = {}; |

The above code defines an object arg1 such that expressions that involve arg1 create lambda functions. For instance, we can now do this:

int main()

{

int data[] = {0,1,2,3,4,5,6,7,8,9};

// Compute the squares

std::transform(data, data+10, data, arg1 \* arg1);

// Write the result to std::cout

std::for\_each(data, data+10, std::cout << arg1 << ' ');

}

The above code displays the following:

0 1 4 9 16 25 36 49 64 81

The expressions arg1 \* arg1 and std::cout << arg1 << ' ' define “functions” that accept one argument and do something to the argument; the passed argument is substituted for arg1 everywhere in the expression and then the expression is evaluated according to the normal rules for C++ expressions. I say “functions” because we all know they’re not really functions, they’re *function objects*—objects of a type that has an operator() member function. In our code, that member function is defined on line 52.

***Standard ML and Proto***

*Functions in Standard ML accept only one value*[*1*](http://cpp-next.com/archive/2010/11/expressive-c-trouble-with-tuples/#fn:ML)*. To pass more, you have to put them in a tuple. As we’ll see, Proto has a similar restriction and the solution is the same.*

The operator() on line 52 only accepts one argument. *All* lambda functions created this way only accept one argument. That sucks! Ordinary C++ functions don’t have that limitation, and lambdas should be as much like ordinary C++ as possible. We need …

**More Lambda Arguments!**

If we want to handle more than one argument, some changes are necessary.

1. We need more overloads of lambda\_expr::operator() that take more arguments.
2. We need more placeholders in addition to arg1 to stand in for those extra arguments.
3. We’ll also pretty clearly need to change our Lambda algorithm to wire together the extra arguments with the extra placeholders, but it’s not clear yet how. *<Hint: we’ll be using functional composition from the last article. (Oooh, foreshadowing.)>*

This is my To-Do list for the rest of the article. (1) and (2) should be pretty easy; (3) is the tricky one. Let’s start at the top, with (1).

**1. More Overloads of lambda\_expr::operator()**

The first order of business is to change lambda\_expr::operator() to accept a variable number of arguments. The old lambda\_expr::operator() accepted only one argument and passed it directly to the Lambda algorithm as the state argument. We’re going to have lots of arguments now, and still only one state argument. So let’s stick them all in a tuple and pass that instead. The code below demonstrates how:

***Tuples***

*Tuples are a generalization of std::pair with a variable number of elements. They come standard in C++0x and TR1. They’re also in Boost with a similar (but not identical) interface. I’ll be using Boost’s Tuple library because it works everywhere, being sure to point out where it differs from the standard.*

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | template<typename ProtoExpr>  struct lambda\_expr  {  // ...    // one-argument operator() for evaluating unary lambdas  template<typename A1>  typename boost::result\_of<Lambda(lambda\_expr const &, boost::tuple<A1 const &> const &)>::type  operator()(A1 const & a1) const  {  boost::tuple<A1 const &> const state(a1);  return Lambda()(\*this, state);  }    // two-argument operator() for evaluating binary lambdas  template<typename A1, typename A2>  typename boost::result\_of<Lambda(lambda\_expr const &, boost::tuple<A1 const &, A2 const &> const &)>::type  operator()(A1 const & a1, A2 const & a2) const  {  boost::tuple<A1 const &, A2 const &> const state(a1, a2);  return Lambda()(\*this, state);  }    // ... more overloads for additional arguments  }; |

*Already we can see what a mess this is going to become. To handle N arguments, we need N overloads of operator(), and it’s mostly repetitious boilerplate. The above code is much nicer in C++0x.*

*Here is the C++0x version of above overload set using a variadic function template for reduced code duplication, rvalue references for proper handling of temporary objects, and unified initialization syntax of “state” just for fun.*

*template<typename... A>*

*typename std::result\_of<Lambda(lambda\_expr const &, std::tuple<A &&...> const &)>::type*

*operator()(A &&... a) const*

*{*

*std::tuple<A &&...> const state{std::forward<A>(a)...};*

*return Lambda()(\*this, state);*

*}*

Lines 11 and 20 above initialize a tuple with the lambda arguments and pass it to the Lambda algorithm. The gnarly return types look worse than they really are. They read: “What type of object does the Lambda algorithm return if I pass it an object of type lambda\_expr and a suitably initialized tuple?” We’re careful everywhere to put const and & in the right place to make it const-correct and to avoid needless copying.

I won’t show it here, but we’ll also need to add *N* partial specializations of the lambda\_expr::result template so that lambda expressions are valid TR1 function objects. (You *so* can’t wait for variadic templates, am I right?) Check the [final solution](http://cpp-next.com/wp-content/uploads/2010/11/07-mini-lambda.txt) for the details.

Item (1) on our To-Do list is done. On to (2).

**2. Mo’ Placeholders**

Arg1 is lonely. Let’s give it friends arg2 and arg3. I’m going to do it in a curious way that will make sense later when its time for item (3).

First, let’s do away with arg1\_tag and arg1\_term and replace them with something more general: class templates for placeholders that are parameterized on an argument *index*:

// A (mostly) empty type that can be used to define lambda

// placeholders. The template parameter is a compile-time

// integer that represents the placeholder number.

template< typename Index >

struct argN\_tag

{

// requires that Index models IntegralConstant

typedef Index index;

};

// A helper for creating placeholder terminals

template< typename Index >

struct argN\_term

: proto::terminal< argN\_tag< Index > >

{};

The template parameter Index is an integer that has been turned into a type. Integral type wrappers come standard in C++0x and TR1 and go by the unwieldy name std::integral\_constant. The [Boost MPL](http://www.boost.org/libs/mpl) has them too, where they have names like mpl::int\_, mpl::long\_, etc. I could use either; they mean the same thing. As usual, I’ll stick to what’s in Boost.

Here is how we define the argument placeholders:

typedef mpl::int\_<0> nil\_t; // or std::integral\_constant<int, 0>

typedef mpl::int\_<1> one\_t; // etc.

typedef mpl::int\_<2> two\_t;

// Define the argument placeholders:

lambda\_expr< argN\_term< nil\_t >::type > const arg1 = {};

lambda\_expr< argN\_term< one\_t >::type > const arg2 = {};

lambda\_expr< argN\_term< two\_t >::type > const arg3 = {};

OK! We’ve checked (1) and (2) off our To-Do list. We can create ternary lambda expressions like arg1 + arg2 + arg3, and the resulting object actually has an operator() member that accepts three arguments. But if we actually try to *use* that member, we’ll find heaps of trouble; we haven’t yet wired up the placeholders to the tuple elements. Onward.

**3. Wiring Placeholders To Tuple Elements**

The Lambda algorithm is the nerve center of our lambda library. It evaluates lambda expressions, substituting actual arguments for placeholders. Now that we’re passing a tuple of arguments to Lambda, it needs to change to accommodate that fact. Let’s look again at the implementation of Lambda. The substitution of actual arguments for placeholders is handled by line 25:

**Actual Proto code:**

|  |  |
| --- | --- |
| 19  20  21  22  23  24  25  26  27  28  29 | // A Proto algorithm for evaluating lambda  // expressions.  struct Lambda  : proto::or\_<  // When evaluating the arg1 terminal,  // return the state.  proto::when< arg1\_term, proto::\_state >  // Otherwise, do the "default" thing.  , proto::otherwise< proto::\_default< Lambda > >  >  {}; |

**Equivalent pseudo-code:**

|  |  |
| --- | --- |
| 19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | // PSEUDO-CODE equivalent for the Lambda  // algorithm to the left.  auto Lambda( auto expr, auto state )  {  // When evaluating the arg1 terminal,  // return the state.  if ( expr matches? arg1 ) return state;  // Otherwise, do the "default" thing.  if ( expr matches? ( \_ + \_ ) )  return Lambda( child<0>(expr), state )  + Lambda( child<1>(expr), state );  if ( expr matches? ( \_ - \_ ) )  return Lambda( child<0>(expr), state )  - Lambda( child<1>(expr), state );  // ...  } |

*If the above pseudo-code didn’t help and Lambda is still a mystery, check out* [*A Lambda Library In 30 Lines (Part 1)*](http://cpp-next.com/archive/2010/10/expressive-c-expression-extension-part-one/)*, which hopefully should clear it up.*

The pseudo-code to the right should give a flavor for what the Proto algorithm to the left is doing. (I’ll have more to say below about the weird “matches?” psuedo-operator.) Line 25 (on the left) reads as follows: each arg1 terminal is replaced with the state. Just as arg1 is a kind of placeholder, so too is proto::\_state. It stands in for and receives Lambda‘s state argument. (Just how Proto manages to ferry the state argument through all the chains of invocations and gets it to appear—as if by magic—in the right place at the right time is left as a mystery for now.)

Now that the state argument is a tuple, we need to change Lambda. For placeholders, Lambda should *index* into the tuple to return the correct argument; arg1 should be replaced with the 0th tuple element, arg2 should return the 1st, etc. Or, more generally, in pseudo-code:

|  |  |
| --- | --- |
| 19  20  21  22  23  24  25  26  27  28 | // PSEUDO-CODE equivalent for the Lambda  // algorithm to the left.  auto Lambda( auto expr, auto state )  {  // When evaluating the arg1 terminal,  // return the state.  if ( expr matches? argN ) return boost::get<N-1>(state);  // Otherwise, do the "default" thing.  // ...  } |

Above is the hand-wavy pseudo-code that expresses our intent: match the Nth placeholder and return the (N-1)th element of the tuple. Boost::get is a function that retrieves an element from a tuple. We want to get the (N-1)th element of that tuple because arg1 corresponds to the 0th element in the tuple, and so on.

This part gets tricky, so first I’m going to show what the actual code looks like and then describe how I got there. So without further ado, here is the new Lambda algorithm (dropping proto:: qualifications from here on out to aid readability):

|  |  |
| --- | --- |
| 19  20  21  22  23  24  25  26  27  28  29 | // A Proto algorithm for evaluating lambda  // expressions.  struct Lambda  : or\_<  // When evaluating the argN terminal,  // return the N-1 element of the state tuple.  when< argN\_term< \_ >, tuple\_get(\_value(\_), \_state) >  // Otherwise, do the "default" thing.  , otherwise< \_default< Lambda > >  >  {}; |

Undoubtedly, this code makes no sense. I haven’t said yet what \_ and \_value are (they’re from the proto:: namespace), nor shown the implementation of tuple\_get. But I aver that this code is valid and works like the pseudo-code above. That means line 25 correctly handles *all* placeholders, not just arg1. It raises two questions:

1. How is it that argN\_term< \_ > correctly “matches” all (and only) the placeholders?
2. How does tuple\_get(\_value(\_), \_state) find and return the correct tuple element?

My use of the made-up “matches?” operator in my pseudo-code might have piqued your curiousity, especially if you’re familiar with a functional language, so let’s start with (1).

**Pattern Matching**

When I say the words, “pattern matching”, what comes to mind? Regular expressions? Bridal patterns? If you’re a functional programmer, the term “pattern matching” should bring you to full attention. Haskell, for instance, can dispatch to different branches by matching a value against a series of patterns. For instance, let’s write a Haskell function to verify the answer to life, the universe and everything:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | *-- Define a function "isTheAnswer" that returns*  *-- True if the argument is 42, False otherwise.*  isTheAnswer 42 **=** True  isTheAnswer **\_** **=** False    main **=** **do** **print** (isTheAnswer 24)  **print** (isTheAnswer 42) |

This program prints:

False True

***P vs. NP***

*It’s easier to verify the answer to life, the universe and everything than it is to compute it.*

It looks like the function isTheAnswer is getting defined twice. It’s really only defined once, but with different patterns. The first pattern is 42, which matches itself. The second pattern is \_, which matches anything; \_ is the so-called wildcard pattern. Here’s how isTheAnswer works: wherever it is invoked, the Haskell runtime performs a test: is the argument 42? If so, the first implementation is selected. If not, it “falls through” to the second implementation. The patterns are tried in the order they appear in the code. (Interestingly, had no pattern matched the argument, it would have been a runtime error, but in our case that’s impossible because the wildcard pattern is acting as a catch-all.)

*Pattern matching in Haskell is a rich and interesting topic. For more information, check out* [*this section on pattern matching*](http://www.haskell.org/~pairwise/intro/section3.html#part1) *in the excellent* [*Haskell Tutorial for C Programmers*](http://www.haskell.org/~pairwise/intro/intro.html)*.*

Why do I bring Haskell into this? Proto does pattern matching, too! Look again at Lambda, and this time I’m going to substitute otherwise on line 27 with the equivalent when construction:

|  |  |
| --- | --- |
| 19  20  21  22  23  24  25  26  27  28  29 | // A Proto algorithm for evaluating lambda  // expressions.  struct Lambda  : or\_<  // When evaluating the argN terminal,  // return the N-1 element of the state tuple.  when< argN\_term< \_ >, tuple\_get(\_value(\_), \_state) >  // Otherwise, do the "default" thing.  , when< \_, \_default< Lambda > >  >  {}; |

Lambda is a function object. To evaluate this function, Proto uses pattern matching. The patterns are argN\_term< \_ > and \_ on lines 25 and 27. Proto tries them in the order they appear in the code, just like in the Haskell example. As you may have guessed by now, \_ is Proto’s wildcard pattern that matches anything.

Recall that the placeholders are now defined as follows:

typedef mpl::int\_<0> nil\_t;

typedef mpl::int\_<1> one\_t;

typedef mpl::int\_<2> two\_t;

lambda\_expr< argN\_term< nil\_t >::type > const arg1 = {};

lambda\_expr< argN\_term< one\_t >::type > const arg2 = {};

lambda\_expr< argN\_term< two\_t >::type > const arg3 = {};

When evaluating Lambda with arg1, Proto matches the type of arg1 against the patterns in Lambda. Ignoring the lambda\_expr wrapper (which has no effect on pattern matching), the type of arg1 is argN\_term< nil\_t >::type. This type gets matched against the pattern argN\_term< \_ >. What happens?

Recall that argN\_term is defined as follows:

template<typename Index>

struct argN\_term

: proto::terminal< argN\_tag<Index> >

{};

Whatever behaviors argN\_term has as a pattern, it inherits from terminal. The terminal class is also where argN\_term gets its nested ::type typedef from. In essence, Proto is really asking the following:

Does the expression type terminal< argN\_tag< nil\_t > >::type match the pattern terminal< argN\_tag< \_ > >?

***Haskell Constructors***

[*Bartosz “Fermat” Milewski*](http://bartoszmilewski.wordpress.com) *has found a truly marvelous proof that the dual nature of proto::terminal is analogous to Haskell data constructors’ secondary use as patterns, but this margin is too small to contain it. Look for it in a future article.*

The answer is yes, but why? Here, terminal is filling two roles. On the one hand, it is being used to generate an expression type (via access to the nested ::type typedef). On the other, it is being used as a pattern that matches expression types. This dual nature of terminal is a bit confusing, but it reduces the number of types needed to learn Proto.

But still we’re left wondering why the type terminal< argN\_tag< nil\_t > >::type matches the pattern terminal< argN\_tag< \_ > >. Well, one is a terminal and the other is a pattern for a terminal. *Match!* Proto probes further: do their *value types* match? That is, does argN\_tag< nil\_t > match argN\_tag< \_ >? Yes, but why? Proto has a set of matching rules that govern how pattern matching proceeds. For example, any terminal value type trivially matches itself (e.g. int matches int), but argN\_tag<nil\_t> and argN\_tag<\_> don’t exactly match.

***Proto Pattern Matching***

*Pattern matching in Proto is a rich topic. It is fully documented* [*here*](http://www.boost.org/doc/libs/1_45_0/doc/html/boost/proto/matches.html)*, but that’s not light reading. We’ll see more examples of Proto patterns over the course of this article series.*

The answer is that Proto has a rule that allows terminal value types of the form T< X > to match types like T< \_ >; i.e., instances of the *same template* where the template parameters themselves match. Proto doesn’t know anything special about argN\_tag, but it can see that argN\_tag<nil\_t> and argN\_tag<\_> conform to this rule, so the match succeeds. And finally it makes sense why I defined argN\_tag the way I did: I wanted to be able to easily pattern-match its type using this rule.

In this way, *all* of the placeholders, arg1, arg2, and arg3, and others we may decide to add later have types that match the pattern argN\_term< \_ >.

**Function Types as Actions**

Now, what about the weird tuple\_get(\_value(\_), \_state) that is used to evaluate lambda placeholders? It looks a lot like an expression, as if we’re making some function calls, doesn’t it? However, it’s used as a parameter to the when template, so it *can’t* be an expression. It is, in fact, a *function type*. The when template interprets function types as function invocations. In other words, when uses function types as a *domain-specific language* for specifying actions. Here’s how to read this Proto-action-as-a-function-type:

1. The type \_ is a placeholder for the expression that just matched.
2. The type \_state is a placeholder for the state.
3. Anything immediately to the left of an opening paren is a TR1-style function object.
4. Anything between parens is an argument to the TR1-style function object.

By applying these rules, proto::when turns the function type tuple\_get(\_value(\_), \_state) into a function object somewhat like the following pseudo-code:

// PSEUDO-CODE equivalent of tuple\_get(\_value(\_), \_state)

auto eval\_argN\_term( auto expr, auto state )

{

return tuple\_get()(\_value()(expr), state);

}

*If you want to know how when converts function types into equivalent function objects, check out the* [*last article*](http://cpp-next.com/archive/2010/11/expressive-c-fun-with-function-composition/)*.*

Above, tuple\_get() and \_value() are default-constructed objects of their respective types. The generation of the function object happens entirely at compile time. The function object is invoked at runtime using the expression that just matched and the current state. In other words, eval\_argN\_term receives either arg1, arg2 or arg3 along with the state tuple.

***Why does \_ receive arg1?***

*In the pattern argN\_term< \_ >, the \_ ends up matching nil\_t. In the action, \_ receives arg1. Why not receive nil\_t, the thing it matched? Consider the following rule from a Proto algorithm:*

*// What gets passed to foo?*

*when< plus<\_,\_>, foo(\_) >*

*In the pattern, \_ shows up more than once, but only once in the action. What does \_ mean in the action? The only sensible thing for Proto to do is to pass the whole expression that matched the pattern (a binary plus expression) to foo. For consistency, \_ in an action always means “the whole expression that just matched”.*

Now all that’s left to explain is the meaning of \_value and tuple\_get. \_value is easy enough: it is a function object that accepts Proto terminals and returns the value inside. The value inside arg1 is an object of type argN\_tag< nil\_t >, so that’s what gets passed to tuple\_get along with the tuple. And now the job of tuple\_get should be clear: it must return the correct element of a tuple given the tuple and an instance of argN\_tag.

***Most Confusing Code Ever?***

*After having a Proto action like tuple\_get(yadda(yadda), yadda) explained to him, a colleague of mine described it as the single most confusing piece of code he’d ever seen. It looked like something, but then he found out it was not that thing, but then it turned out it really was.*

*In the last article I made a case for function types being a little domain-specific language for function composition. So it is with Proto. Once seen as a separate little language, Proto actions are easier to grok. At some point you give up trying to reason it through as C++ and just see it as Proto*.

**Details, Details**

Compared with the heady pattern matching and function composition stuff, the final task seems rather prosaic: write a function object named tuple\_get that, when passed an instance of argN\_tag and a tuple, returns the correct element of the tuple. We just need to wrap the correct boost::get function call. What follows is a TR1-style function object that does just that, in all its prosaic glory. Explanation to follow.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | // A TR1-style function object that accepts an argN\_tag  // instance and a tuple and returns the proper tuple element.  struct tuple\_get : proto::callable  {  // Nested result template for return type calculation  template<typename Sig> struct result;    template<typename This, typename ArgNTag, typename Tuple>  struct result<This(ArgNTag const &, Tuple const &)>  {  // ArgNTag is an instance of argN\_tag, which has a nested index typedef  typedef typename ArgNTag::index N;  // boost::tuples::element is a trait that returns the type of the Nth element  typedef typename boost::tuples::element<N::value, Tuple>::type type;  };    template<typename ArgNTag, typename Tuple>  typename result<tuple\_get(ArgNTag const &, Tuple const &)>::type  operator()(ArgNTag const &, Tuple const & tup) const  {  typedef typename ArgNTag::index N;  // boost::get is a function that returns the Nth element of a tuple.  return boost::get<N::value>(tup);  }  }; |

As usual with TR1-style function objects, it’s a little hard to see the intent through all the syntactic noise, but there it is on line 23: a call to boost::get that fetches an element from a tuple.

Some details bear mentioning. Tuple\_get accepts as its first argument an instance of argN\_tag, which—if you look waaaaay back to where we defined it above—has a nested typedef called index. It’s a type-wrapped integer representing a tuple index. Type-wrapped integers like mpl::int\_ and std::integral\_wrapper have a nested static constant called value that (unsurprisingly) is the value of the wrapped integer. So mpl::int\_<1>::value is 1. But it’s a *compile-time constant*, so we can use it as a template parameter to boost::get, which is important. Anyway, that’s where ::value comes from on lines 14 and 23.

Line 9 is the partially specialized nested result template, the idiomatic way to compute the return type of tuple\_get::operator(). This TR1 result\_of song and dance should be feeling vaguely familiar by now. Line 14 has the compile-time equivalent of the boost::get call. Boost::tuples::element is a trait that takes a compile-time integer and a tuple type, and “returns” the type of the element stored at that index.

**Boost Tuples vs. C++0x Tuples**

The template in Boost called boost::tuples::element is known as std::tuple\_element in TR1 and C++0x. That’s the only practical difference.

The only other noteworthy thing in the tuple\_get implementation is the inheritance from proto::callable. I haven’t shown enough about Proto actions to explain why this is necessary. For now, I’ll just say this: all TR1-style function objects used to compose actions using function types (like get\_tuple(yadda(yadda), yadda)) must inherit from callable directly or indirectly. It helps Proto make sense of the function type. But I’ll spare you and defer the full explanation for another article.

**Summing Up**

Here, once again, is the new and improved Lambda algorithm:

|  |  |
| --- | --- |
| 19  20  21  22  23  24  25  26  27  28  29 | // A Proto algorithm for evaluating lambda  // expressions.  struct Lambda  : or\_<  // When evaluating the argN terminal,  // return the N-1 element of the state tuple.  when< argN\_term< \_ >, tuple\_get(\_value(\_), \_state) >  // Otherwise, do the "default" thing.  , when< \_, \_default< Lambda > >  >  {}; |

Now that we know what’s going on, we can appreciate how the parts work together: compile-time function dispatch using pattern matching, function composition using function types, and TR1-style function objects. Each little piece, say tuple\_get, is not hard to write or grok in isolation. The pieces snap together with a concise syntax that feels natural (or will come to feel natural if you stick with it). In total, Lambda is quite a powerful little beastie; it can evaluate arbitrarily complicated lambda expressions with any number of argument placeholders. All that in 6 lines of code. (OK, we had to write a few to set up those 6 lines, but work with me.)

[Click here to view the complete solution](http://cpp-next.com/wp-content/uploads/2010/11/07-mini-lambda.txt).

I’ll leave you with a simple program that uses our lambda library and the binary lambda expression arg1 + arg2 with the two-input-sequence std::transform algorithm to—what else?—generate the Fibonacci numbers. Enjoy.

#include <algorithm>

int main()

{

int data[10] = {0,1};

// generate the Fibonacci sequence

std::transform(data, data+8, data+1, data+2, arg1 + arg2);

// write each element to std::cout

std::for\_each(data, data+10, std::cout << arg1 << ' ');

}

**Acknowledgements**

Thanks to [Bartosz Milewski](http://bartoszmilewski.wordpress.com), [Walter Bright](http://www.walterbright.com), and [Andrei Alexandrescu](http://erdani.com) for their valuable feedback about this post.

1. See [Tuples as arguments in Standard ML](http://en.wikibooks.org/wiki/Standard_ML_Programming/Types#Tuples_as_arguments). [↩](http://cpp-next.com/archive/2010/11/expressive-c-trouble-with-tuples/#fnref:ML)

Posted Monday, November 22nd, 2010 under [Boost](http://cpp-next.com/archive/category/boost/), [Functional Programming](http://cpp-next.com/archive/category/functional-programming/).