Posted November 4, 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++: Fun With Function Composition**

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
7. [Expressive C++: Trouble With Tuples](http://cpp-next.com/archive/2010/11/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++. This time around we’re going to take a bit of a diversion from EDSLs to talk about [function composition](http://en.wikipedia.org/wiki/Function_composition_%28computer_science%29). C++ does it differently from languages with [higher-order functions](http://en.wikipedia.org/wiki/Higher-order_function) like Haskell. We’ll see how C++ fares at function composition against Haskell (poorly) and how C++0x will bring partial relief. We’ll sketch a concise notation for composing functions in C++ and hint at how it can simplify the manipulation of expression trees in Proto. Along the way, we’ll shed some light on TR1′s result\_of utility.

**What is Function Composition?**

*Why Is Function Composition Interesting?*

*Function composition promotes code reuse. A generic library of functions would never ship a function square\_and\_add\_one, but it might ship square and inc, which is all you need if you can easily compose them.*

The idea behind function composition is really very simple: given two functions, create a new function that has the effect of applying one to the result of applying the other. These sorts of operations on functions are much easier in functional languages. For instance, let’s define two trivial functions, inc and square and compose them to make square\_and\_add\_one. The language I’m using for this is Haskell. If you don’t know Haskell, that’s ok. I’ll explain it below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | *-- Define a function "inc" that increments its argument*  inc x **=** x **+** 1    *-- Define a function "square" that squares its argument*  square x **=** x **\*** x    *-- Define a function "square\_and\_add\_one" that squares its*  *-- argument and increments it by one*  square**\_**and**\_**add**\_**one **=** (inc **.** square)    *-- This prints "10"*  main **=** **do** **print** (square**\_**and**\_**add**\_**one 3) |

For the C++ programmers, it’s probably easiest to understand this Haskell code by looking at the equivalent C++:

|  |  |
| --- | --- |
| 13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | #include <iostream>    template<typename T>  T inc(T x) { return x + 1; }    template<typename T>  T square(T x) { return x \* x; }    template<typename T>  T square\_and\_add\_one(T x) { return inc(square(x)); }    int main()  {  std::cout << square\_and\_add\_one(3) << std::endl;  } |

*Why am I even talking about Haskell? Isn’t this an article series devoted to embedded domain-specific languages in C++? Yes, but it turns out EDSLs in C++ (and* [*template metaprogramming*](http://en.wikipedia.org/wiki/Template_metaprogramming) *in general) share a lot of common ground with functional programming. It’s very handy to reason about these things in a language that isn’t as ill-suited to the task at hand as C++ templates.*

These two programs are very similar. Of particular interest is the fact that in C++ we had to go out of our way to say that these functions were templates. In Haskell, we get the equivalent for free. Haskell is strongly typed like C++, but it has a powerful type system that lets its compiler figure types out. So, for instance, on line 2, the Haskell compiler knows that inc is a function that takes and returns a T even though nobody told it.

The definitions of square\_and\_add\_one are where the two solutions diverge. You’re probably familiar with how type deduction works in C++ function templates (like how the T on line 16 is deduced to be int from line 26). And if the C++ compiler can do that, the Haskell compiler can do it too. But the Haskell definition of square\_and\_add\_one is a whole other ball of wax. We didn’t even have to say how many arguments it takes! Why?

The answer is to look at the definition of square\_and\_add\_one on line 9. Notice the use of the infix . operator. That’s Haskell’s function composition operator. Haskell knows to take the expression (inc . square) and build a new function that is the equivalent of inc (square x). (We need the parenthesis around square x because without them, Haskell would try to pass both square and x to inc as arguments, which would be an error.) Haskell can look at inc (square x) and figure out its type based on the types of inc and square: it’s also a function that takes and returns a T. Since the compiler can figure it out, we don’t have to say it.

In contrast, look at the C++ definition of square\_and\_add\_one: you have to say up front in the function signature that it *also* is a function that takes and returns a T, even though the compiler technically *could* figure that out based on the function body. Bummer.

**Function Composition in a Library**

The Haskell function composition operator is actually not part of the core language. It’s a pure library extension. Not to be outdone, we C++ programmers should also be able to define function composition in a library, right? That is, we want to define a reusable component that, given any two unary (single-argument) functions F1 and F2, defines a third function C that behaves just like F1(F2(x)). To be fully general, F1 and F2 are allowed to be *polymorphic*[1](http://cpp-next.com/archive/2010/11/expressive-c-fun-with-function-composition/#fn:polymorphic) (they can operate on data of any type), just as in the Haskell and C++ examples above.

***Higher-Order Functions***

*Function composition is an example of a* [*higher-order function*](http://en.wikipedia.org/wiki/Higher-order_function)*: a function that accepts other functions as arguments. You’re probably already familiar with higher-order functions from the STL. For example, std::accumulate is a higher-order function because it accepts a function object as an argument.*

What does such a library solution look like? Let’s try to generalize the function composition being done in square\_and\_add\_one above. Instead of hard-coding square and inc, let’s pass them as arguments instead. Great idea, but it immediately falls flat. How do you pass a *function template* like square to another function? You can’t, because a function template is a non-entity: it has no address and occupies no space. You can’t even use it as template parameter! The only thing you can do with a function template is use it to instantiate functions. That is, &square does not compile, but &square<int>—which instantiates square with int and takes the address of the resulting function—does. Function templates are not [first-class objects](http://en.wikipedia.org/wiki/First-class_object). To add insult to injury, square<int> is no longer polymorphic: it only operates on integers. In contrast, Haskell functions are both first-class objects and polymorphic, which makes function composition in that language much easier.

Pretty quickly we’ve run up against a hard limitation in C++: to compose polymorphic functions, we can’t use functions! Ouch.

**Polymorphic Function Objects**

We’re not dead yet. The STL’s higher-order functions like std::accumulate show us a way forward: wrap the function in an *object*. By defining a class with an operator() member function, we get first-class objects that behave like functions. Those objects have all the goodness we expect from first-class objects: we can pass them, return them, store them, and copy them to our heart’s content. We can also compose them.

*It is somewhat ironic that the only way to compose functions in C++ is to give up on functions entirely and use function objects instead.*

I should point out that the STL’s function objects like std::plus are *not* polymorphic. That is, you can create an object of type std::plus<int>, but it’s operator() only takes and returns integers. We can make plus polymorphic by moving the parameterization from the class to the member function. That is, instead of making the plus class a template, we make its operator() a template instead. An object of this plus type would behave just like a polymorphic function.

Let’s rewrite inc and square as polymorphic function objects, and define a composed\_fn template that we can use to compose the two to create square\_and\_add\_one. The code is explained below:

|  |  |
| --- | --- |
| 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  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48 | #include <iostream>    // A polymorphic function object type  struct inc\_t  {  template<typename T>  T operator()(T x) const { return x + 1; }  };    // Instances of function objects behave like polymorphic functions  inc\_t const inc = inc\_t();    struct square\_t  {  template<typename T>  T operator()(T x) const { return x \* x; }  };    square\_t const square = square\_t();    // Here is a function object that composes two other  // function objects. It behaves like Haskell's infix . operator.  template<typename Fn1, typename Fn2>  struct composed\_fn  {  explicit composed\_fn(Fn1 f1 = Fn1(), Fn2 f2 = Fn2())  : fn1(f1), fn2(f2)  {}    template<typename T>  T operator()(T x) const // PROBLEM HERE  {  return fn1(fn2(x));  }    Fn1 fn1;  Fn2 fn2;  };    // Compose two functions!  typedef composed\_fn<inc\_t, square\_t> square\_and\_add\_one\_t;  square\_and\_add\_one\_t const square\_and\_add\_one = square\_and\_add\_one\_t();    int main()  {  // This prints "10"  std::cout << square\_and\_add\_one(3) << std::endl;  } |

This solution is lacking in one important respect, but at least it works. We can use the composed\_fn template to compose two “functions”. The line inc\_t const inc = inc\_t() is declaring a global object called inc and initializing it with inc\_t(), which is a default-constructed instance of type inc\_t.

*To solve the return type deduction problem using decltype and the new-style function declaration syntax in C++0x, do the following:*

template<typename Fn1, typename Fn2>

struct composed\_fn

{

Fn1 fn1;

Fn2 fn2;

explicit composed\_fn(Fn1 f1 = Fn1(),

Fn2 f2 = Fn2())

: fn1(f1), fn2(f2)

{}

template<typename T>

auto operator()(T x) const

-> decltype(fn1(fn2(x)))

{

return fn1(fn2(x));

}

};

The problem with this solution happens on line 31. The composed\_fn template is supposed to be generic: take two function objects and compose them. However, it is making a huge assumption that greatly reduces its genericity: that when passed an object of type T, fn1(fn2(x)) also returns an object of type T. Although it happens to be the case in our toy example, it is by no means guaranteed. If you’re using a C++0x compiler, the problem is solvable with decltype and the new-style function declaration syntax (see sidebar), but for those of us living in the present, some hoop-jumping is necessary. The composed\_fn template needs a way to ask the types Fn1 and Fn2, “When I pass you an object of type T, what is the type of the object you return?” Enter the TR1 result\_of protocol.

**The TR1 result\_of Protocol**

In Haskell, the compiler just knows the type of functions and can propogate types automatically when functions are composed. We’re not so lucky in C++, which is why TR1 included a utility called result\_of and an associated protocol for declaring and calculating function object return types. We can solve our function composition problem if we change our function objects to follow the protocol, as follows:

// Redefine inc to be a function object

// that follows the TR1 result\_of protocol

struct inc\_t

{

// Nested result templates are for calculating

// the return type.

template<typename Signature>

struct result;

template<typename This, typename T>

struct result<This(T)> { typedef T type; };

template<typename T>

T operator()(T x) const { return x + 1; }

};

inc\_t const inc = inc\_t();

*There is another way to satisfy the result\_of protocol—by defining a nested result\_type typedef, making standard C++03 functors like std::plus valid—so you never want to access the nested result template or the result\_type typedef directly; either may be missing. That’s the raison d’être of result\_of. Hidden inside is some magic that figures out how a type implements the protocol before using it.*

Now, even without decltype, there’s a way to ask inc\_t what kind of object it will return when passed an int. The answer is: inc\_t::result<inc\_t(int)>::type, which we can tell is an alias for int. (I’ll have more to say about the strange inc\_t(int) type below.) In general you don’t want to access the nested result template directly like this (see sidebar). It’s better to use the result\_of utility, either in namespace std, std::tr1, or boost depending on how far along your C++ implementation is. Since the [Boost](http://boost.org) imlementation works everywhere, I’ll stick to that: boost::result\_of< inc\_t(int) >::type.

***Why is result\_of so complicated?***

*At first blush, the result\_of protocol looks more complicated than it needs to be. After all, wouldn’t this work just as well?*

*struct simple\_fn*

*{*

*// This is NOT the TR1 result\_of protocol. Don't*

*// do it this way:*

*template<typename T>*

*struct result { typedef T type; };*

*template<typename T>*

*T operator()(T x) const;*

*};*

*Admitedly, this nested result template is far simpler than the one in inc\_t. If you want to know what simple\_fn returns when passed an int, you do this: simple\_fn::result<int>::type. Simple, right? This solution generalizes poorly, though. Consider function objects that have several overloaded operator() members, such as:*

|  |  |
| --- | --- |
| *1*  *2*  *3*  *4*  *5*  *6*  *7*  *8*  *9* | *struct weird\_fn*  *{*  *template<typename T>*  *struct result { /\* What the heck goes here??? \*/ };*    *int & operator()(int);*  *int const & operator()(int) const; // overload on constness of \*this*  *float operator()(float, float); // accepts different numbers of arguments*  *};* |

*Lines 6 and 7 overload operator() based on the constness of the weird\_fn object, which is perfectly valid. Also, line 8 defines an operator() that takes a different number of arguments than the first two. The simplified return protocol of simple\_fn can’t handle these cases. With the TR1 result\_of protocol, the types of the three operator() members can be obtained with the following queries:*

*boost::result\_of< weird\_fn(int) >::type // int &*

*boost::result\_of< weird\_fn const(int) >::type // int const &*

*boost::result\_of< weird\_fn(float, float) >::type // float*

*By specializing weird\_fn::result on weird\_fn(int), weird\_fn const(int), and weird\_fn(float, float), we can accomodate all of these cases. (In inc\_t above, we partially specialized the result template on This(T) instead of on inc\_t(T), allowing This to be deduced as either inc\_t or inc\_t const. This is a common shortcut.)*

*In short, the TR1 protocol is designed to accomodate a broad range of function objects.*

**C++ Function Types As A DSL**

We can deduce that boost::result\_of< inc\_t(int) >::type is an alias for int. But wait, let’s think about this a moment. What is inc\_t(int)? It’s a *function type*. If you’re an old-school C programmer, you’re probably more familiar with function pointers like int(\*)(int), which is a pointer to a function that takes and returns an int. If you remove the pointer, you’re left with a function type. It’s like a function signature, but without the function name. You can’t create variables of function type, but that doesn’t keep you from stating the type in code.

If we read the type inc\_t(int) literally, it is the type of a function that takes an int and returns an object of type inc\_t. But nowhere have we defined a function that *returns inc\_t*. Clearly something funny is going on. How do we make sense of the type inc\_t(int)? The answer is that result\_of *interprets* this type in a special way; it reads it as a query: (roughly) what is the type of the expression inc\_t()(int())? (Here inc\_t() and int() are temporary objects constructed in-place via the default constructor.) Result\_of need not actually evaluate the expression to find its type; instead it knows to look inside inc\_t for the result template, because that’s the agreed-upon protocol.

In other words, result\_of uses function types as a **domain-specific language** for specifying function invocations. Outside of result\_of, the type inc\_t(int) means one thing; inside it, it means something else. This DSL is so effective, this subterfuge might have slipped by you. This is a very important point, and we’ll be coming back to it.

**Function Composition With result\_of**

Now that we have a way to declare and compute function object return types, we can finally write a correct composed\_fn implementation. We can even make the interface a little more intuitive by following result\_of‘s lead and using function types as a DSL for describing function invocation:

// An undefined type, used as an argument placeholder in our DSL

struct \_;

template<typename Signature>

struct composed\_fn;

// An improved composed\_fn template that uses the TR1 result\_of

// protocol to declare and compute function return types.

template<typename Fn1, typename Fn2>

struct composed\_fn<Fn1(Fn2(\_))>

{

explicit composed\_fn(Fn1 f1 = Fn1(), Fn2 f2 = Fn2())

: fn1(f1), fn2(f2)

{}

template<typename Signature>

struct result;

template<typename This, typename T>

struct result<This(T)>

{

typedef typename boost::result\_of<Fn2(T)>::type U;

typedef typename boost::result\_of<Fn1(U)>::type type;

};

template<typename T>

typename result<composed\_fn(T)>::type operator()(T x) const

{

return fn1(fn2(x));

}

Fn1 fn1;

Fn2 fn2;

};

// Compose two functions. Use a function type to specify how

// the functions should be composed.

typedef composed\_fn< inc\_t(square\_t(\_)) > square\_and\_add\_one\_t;

square\_and\_add\_one\_t const square\_and\_add\_one = square\_and\_add\_one\_t();

This composed\_fn template is able to compose any two unary polymorphic function objects that implement the result\_of protocol. Notice the use of the function type inc\_t(square\_t(\_)) in the definition of square\_and\_add\_one. We declared a dummy type called \_ to act as a placeholder, and used it in a pseudo-function-call-expression made up of nested function types. More than mere cuteness, this interface has the benefit of making it virtually impossible to get confused about which function gets invoked first.

Finally, after dozens of lines of code (and many more hidden away in boost::result\_of) we have matched what Haskell did in 5 lines. *<sigh>*

**Generalizing Function Types as a DSL**

Using function types to represent function calls is a handy trick. Now that we know it, we can extend it to make our composed\_fn template more powerful with little extra effort. Using function types, composed\_fn can glom together *any number of functions* with a simple and intuitive interface. For instance, what would you expect the following to do?

// What do you think this should do?

composed\_fn< inc\_t(square\_t(inc\_t(square\_t(\_)))) > fun;

int i = fun(3);

If you’re like me, you’d expect it to be equivalent to:

int i = inc(square(inc(square(3))));

assert( i == 101 );

That doesn’t seem like too much of an intellectual leap. Our composed\_fn template doesn’t do that yet, but it’s not hard to add using recursive function composition and one itsy, bitsy trick. Continue reading for the explanation.

|  |  |
| --- | --- |
| 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  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48 | // An undefined type, used as an argument placeholder in our DSL  struct \_;    template<typename Signature>  struct composed\_fn;    // An improved composed\_fn template that uses the TR1 result\_of  // protocol to declare and compute function return types.  template<typename Fn1, typename F>  struct composed\_fn<Fn1(F)>  {  // Recursively compose all nested function types to build  // a composed function object out of them.  typedef composed\_fn<F> Fn2;    explicit composed\_fn(Fn1 f1 = Fn1(), Fn2 f2 = Fn2())  : fn1(f1), fn2(f2)  {}    template<typename Signature>  struct result;    template<typename This, typename T>  struct result<This(T)>  {  typedef typename boost::result\_of<Fn2(T)>::type U;  typedef typename boost::result\_of<Fn1(U)>::type type;  };    template<typename T>  typename result<composed\_fn(T)>::type  operator()(T x) const  {  return fn1(fn2(x));  }    Fn1 fn1;  Fn2 fn2;  };    // This specialization ends the recursion begun on line 14.  // "composed\_fn<Fn(\_)> fn; fn(3);" is equivalent to "Fn fn; fn(3);"  template<typename Fn>  struct composed\_fn<Fn(\_)> : Fn {};    // TRICKSY!!! Read below for why this specialization is necessary.  template<typename Fn>  struct composed\_fn<Fn \*> : composed\_fn<Fn> {}; |

With this definition of composed\_fn, you can crazily compose function objects to your heart’s content. As promised above, you can do this sort of nuttiness:

// Whole lotta function composition goin' on:

composed\_fn< inc\_t(square\_t(inc\_t(square\_t(\_)))) > fn;

// prints 101 as expected

std::cout << fn(3) << std::endl;

The only tricky part is that function types “decay” to function *pointer* types in some usages. Because you can’t pass a function to a function, when the compiler sees an otherwise nonsensical type like Fn1(Fn2(\_)), it thinks, “Oh, this person probably meant Fn1(Fn2(\*)(\_)) *and actually compiles it that way*. It’s called *type decay*, and it happens also with array types (they decompose into pointers, too). We need to accomodate that strangeness by adding a specialization for pointers on line 48. It just strips the pointer and forwards its implementation on.

**What’s The Big Deal?**

OK, so you can create ridiculous types like “composed\_fn< inc\_t(square\_t(inc\_t(square\_t(\_)))) >“. What’s the use of that? Granted, with inc and square it’s not very exciting. But consider instead polymorphic function objects called first and second that return the first and second members of a std::pair, for instance:

// Given a std::pair p, first(p) return p.first (implementation not shown)

first\_t const first = first\_t();

// ... and second(p) returns p.second

second\_t const second = second\_t();

Now imagine a big tree-like data structure composed entirely of pairs of pairs of pairs, etc. What do you think about a type like this:

// What does this type signify?

composed\_fn< first\_t(second\_t(second\_t(\_))) >

It describes a function object that navigates that tree and returns a particular element, right?

This type has some interesting properties. Not only can we use this type to do a computation at compile time (what is the type of that nested element?), but we can also use it to perform an analogous computation at runtime (fetch me that element). It’s compile-time and runtime computation, tied up in a neat little package. It’s concise, and (like a good little DSL) it does what it says.

What’s more, functions composed this way are very efficient. Nothing is hidden from the optimizer. What appears like a terrific number of function calls can be fully inlined. Applying composed\_fn< first\_t(second\_t(second\_t(\_))) > to a pair p is exactly equivalent to saying p.second.second.first. The advantage of the composed function is that it is useful both at compile time and at runtime; the expression is only useful at runtime.[2](http://cpp-next.com/archive/2010/11/expressive-c-fun-with-function-composition/#fn:decltype)

***Function Composition and Boost.Proto***

*Why am I hammering on about function composition? What is the connection with domain-specific languages and Boost.Proto? If you’ve been following this article series, you know that Proto turns C++ expressions into tree objects that are much like pairs of pairs. We can use function types as a very concise way to describe computations on expression trees: we can navigate them, pull out elements, apply transformations, compute results, build new trees, whatever. To hook into this mechanism, all we need to know is how to write a TR1-style function object and how to compose those function objects using function types. And now we know.*

**Conclusions And What’s To Come**

Haskell programmers who have made it this far are probably rolling their eyes like the possessed. We *still* haven’t done anything they couldn’t do in a fraction of the code. But buck up C++ programmers! Unlike Haskell, C++ is a multi-paradigm programming language, giving you many ways to express your solution. And the fact that we can add functional constructs to C++ in a library is a testament to its strength. (The fact that we *have* to is a testament of a different sort, alas.)

Programming C++ at runtime can be imperative, object-oriented, functional, generic, whatever. Use the paradigm that fits your problem and your mood. But programing C++ at compile-time (using templates), is purely functional—no mutable state, please! And if you want to do *both at once*—like with our composed\_fn template—you have to settle for the intersection of the compile-time and runtime paradigms: functional programming. For that reason, getting your head around a language like Haskell that gets FP right is worth the time investment.

In the next article in the series, we’ll see how to use the function types DSL we developed in this article to extend the Lambda Proto algorithm from the last two articles. When we’re done, the little lambda library we’ve been developing will be far more useful.

Until then….

**Acknowledgements**

Thanks to [Bartosz Milewski](http://bartoszmilewski.wordpress.com) and [David Abrahams](http://www.boostpro.com) for their valuable feedback about this post.

1. Despite the strong associations with virtual functions that the term “polymorphic” conjures in the minds of C++ programmers, it actually refers to a function that can operate on data of many different types. That includes your run-of-the-mill OO-style dynamic polymorphism (virtual functions) as well as static polymorphism (templates). [↩](http://cpp-next.com/archive/2010/11/expressive-c-fun-with-function-composition/#fnref:polymorphic)
2. Unless, of course, you have decltype. C++0x will make this all so much easier. [↩](http://cpp-next.com/archive/2010/11/expressive-c-fun-with-function-composition/#fnref:decltype)

Posted Thursday, November 4th, 2010 under [Boost](http://cpp-next.com/archive/category/boost/), [Functional Programming](http://cpp-next.com/archive/category/functional-programming/).