Posted September 23, 2010 by [Eric Niebler](http://ericniebler.com), under [Boost](http://cpp-next.com/archive/category/boost/)

From <http://cpp-next.com/archive/2010/09/expressive-c-why-template-errors-suck-and-what-you-can-do-about-it/>

## Expressive C++: Why Template Errors Suck[[1]](#footnote-1) and What You Can Do About It

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**
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](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 part 3 of Expressive C++, a series of articles devoted Embedded Domain-Specific Languages (EDSLs)[1](http://cpp-next.com/archive/2010/09/expressive-c-why-template-errors-suck-and-what-you-can-do-about-it/#fn:8)) and Boost.Proto, a library for implementing them[[2]](#footnote-2) in C++. The title of this article is intentionally provocative[[3]](#footnote-3) to give me the creative license[[4]](#footnote-4) I need to get this righteous rage[[5]](#footnote-5) out of my system, lay some much-deserved blame[[6]](#footnote-6), and—after my catharsis[[7]](#footnote-7)—offer some constructive suggestions for improving the situation. You might be surprised at where I direct my ire[[8]](#footnote-8), and also pleased to know that whether you’re a library author or a user, there are things you can do to help to improve the situation.

Eventually, we’ll bring the discussion back to EDSLs and apply my recommendations to the simple string formatting routine we developed in the [last installment](http://cpp-next.com/archive/2010/09/expressive-c-playing-with-syntax/). By the end of the article, you’ll know how to syntax-check an EDSL by defining its grammar, validate that an expression matches the grammar, and issue a short and meaningful diagnostic if it doesn’t.

### Template Errors: A Sad State Of Affairs[[9]](#footnote-9)

Breaking news from the C++ meta-verse[[10]](#footnote-10)[2](http://cpp-next.com/archive/2010/09/expressive-c-why-template-errors-suck-and-what-you-can-do-about-it/#fn:2):

TEMPLATE ERROR MESSAGES ARE TERRIBLE!

<yawn*[[11]](#footnote-11)*>. That’s no news to anyone who’s used C++ in the last 10 years or so. Even simple misuses of template libraries can lead to 100′s of Kb of compiler spew[[12]](#footnote-12). Who’s to blame? Take your pick[[13]](#footnote-13): library authors, compiler vendors, or the C++ standardization committee? They’ve each felt their share of the heat[[14]](#footnote-14). The key selling point[[15]](#footnote-15) of [C++0x concepts](http://en.wikipedia.org/wiki/Concepts_%28C%2B%2B%29) ([R.I.P.](http://cpp-next.com/archive/2009/08/what-happened-in-frankfurt)[[16]](#footnote-16)) was improved template errors. And one of the key selling points of [clang](http://clang.llvm.org), an exciting, new C/C++ compiler in active development, is better error messages. But in my personal experience as a library developer, I believe this problem begins at home[[17]](#footnote-17) : with poorly designed and implemented template libraries.

Library techniques for improved compile-time error detection and reporting have existed for a while[[18]](#footnote-18), but they’re not commonly known or widely used.[3](http://cpp-next.com/archive/2010/09/expressive-c-why-template-errors-suck-and-what-you-can-do-about-it/#fn:10) If folks[[19]](#footnote-19) only knew how much better the world would be if these techniques were consistently applied, we wouldn’t settle for[[20]](#footnote-20) 100′s of Kbs of compiler spew. We’d be outraged[[21]](#footnote-21).

If I’m not being clear enough, let me say it explicitly, and in a way[[22]](#footnote-22) that’s likely to raise a few eyebrows[[23]](#footnote-23):

Bad template errors are **library bugs** and should be reported as such.

The implication for library users is simple: stop cursing[[24]](#footnote-24) the darkness and start cursing library authors. Well, don’t curse them because they might be me. File bugs[[25]](#footnote-25) instead. Yes, really. (And if you just can’t wait for the bugs to be fixed, switch to clang or install [STLFilt](http://www.bdsoft.com/tools/stlfilt.html).)

So that’s my big advice to library users? Complain? Yes, but complain to the right people. Not on your blog, not to [Reddit](http://www.reddit.com/), but to the author of the offending29 library. And do it by filing a reproducible bug. If we all did that, the library authors would get the message that these problems can and should be fixed

What are the implications for library writers? What could a library author possibly do to fix these so-called “bugs”? And hey[[26]](#footnote-26), why are bad template errors endemic[[27]](#footnote-27) in the first place? Simply put[[28]](#footnote-28): a total lack of parameter validation.[[29]](#footnote-29)

### Remedial[[30]](#footnote-30) Software 101[[31]](#footnote-31)

When you first starting writing code, someone probably told you how important it is to validate parameters at (runtime) API boundaries. Null pointers, out-of-bounds indices, incorrectly escaped URLs—if you fail to check for them, you’ll end up with runtime bugs that hackers can exploit[[32]](#footnote-32). Any programmer worth his salt will tell you this.[[33]](#footnote-33)

But when those same programmers sit down to write a template, many tend to forget this very basic advice and blithely[[34]](#footnote-34) accept user-supplied types without doing any parameter checking at all. The result is a car wreck of epic proportions[[35]](#footnote-35).

Almost all templates place certain requirements on their parameters: that they have certain member functions, nested typedefs, etc. To validate a template parameter would be to check up front that these conditions are met (to the extent it’s possible) and fail with a meaningful diagnostic if they’re not.

Let’s take the [Boost.Spirit](http://boost-spirit.com) example from the [Intro](http://boost-spirit.com) and modify it slightly:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | #include <boost/spirit/home/qi.hpp>    int main()  {  using namespace boost::spirit::qi;  rule<char const \*> expression, term, factor;    expression = term >> \*( ( '+' >> term ) | ( '-' >> term ) ) ;  term = factor >> \*( ( '\*' >> ~factor ) | ( '/' >> factor ) ) ;  factor = uint\_ | '(' >> expression >> ')' | '-' >> factor ;  } |

Can you spot the typo[[36]](#footnote-36) in the code?[[37]](#footnote-37) The resulting 160 Kb of compiler spew is enough to make a sane programmer run screaming[[38]](#footnote-38)[4](http://cpp-next.com/archive/2010/09/expressive-c-why-template-errors-suck-and-what-you-can-do-about-it/#fn:3):

In file included from /home/Eric/boost/org/trunk/boost/spirit/home/qi/char.hpp:14:0, from /home/Eric/boost/org/trunk/boost/spirit/home/qi.hpp:17,

from main.cpp:1:

/home/Eric/boost/org/trunk/boost/spirit/home/qi/char/char\_parser.hpp: In instantiation of ‘boost::spirit::qi::make\_composite<boost::proto::tag::complement, boost::fusion::cons<boost::spirit::qi::reference<const boost::spirit::qi::rule<const char\*> >, boost::fusion::nil>, boost::fusion::unused\_type, void>’:/home/Eric/boost/org/trunk/boost/spirit/home/qi/meta\_compiler.hpp:103:13: instantiated from ...<snip enormous template instantiation backtrace>/home/Eric/boost/org/trunk/boost/mpl/if.hpp:70:41: error: ‘value’ is not a member of ‘boost::spirit::traits::has\_no\_unused<boost::fusion::transform\_view<boost::fusion::cons<boost::spirit::qi::sequence<boost::fusion::cons<boost::spirit::qi::literal\_char<boost::spirit::char\_encoding::standard, true, false>, boost::fusion::cons<boost::spirit::qi::negated\_char\_parser<boost::spirit::qi::reference<const boost::spirit::qi::rule<const char\*> > >, boost::fusion::nil> > >, boos

t::fusion::cons<boost::spirit::qi::sequence<boost::fusion::cons<boost::spirit::qi::literal\_char<boost::spirit::char\_encoding::standard, true, false>, boost::fusion::cons<boost::spirit::qi::reference<const boost::spirit::qi::rule<const c

har\*> >, boost::fusion::nil> > >, boost::fusion::nil> >, boost::spirit::traits::build\_attribute\_sequence<boost::fusion::cons<boost::spirit::qi::sequence<boost::fusion::cons<boos

t::spirit::qi::literal\_char<boost::spirit::char\_encoding::standard, true, false>, boost::fusion::cons<boost::spirit::qi::negated\_char\_parser<boost::spirit::qi::reference<const boost::spirit::qi::rule<const char\*> > >, boost::fusion::nil> > >, boost::fusion::cons<boost::spirit::qi::sequence<boost::fusion::cons<boost::spirit::qi::literal\_char<boost::spirit::char\_encoding::standard, true, false>, boost::fusion::cons<boost::spirit::qi::reference<const boost::spirit::qi::rule<const char\*> >, boost::fusion::nil> > >, boost::fusion::nil> >, boost::spirit::context<boost::fusion::cons<boost::fusion::unused\_type&, boost::fusion::nil>, boost::fusion::vector0<> >, boost::mpl::identity, const char\*>::element\_attribute, boost::fusion::void\_> >’

The mistake is that the definition of the term rule is invalid, but it’s not easy to infer that from this mountain of spew.

The problem of error detection and reporting is particularly acute[[39]](#footnote-39) in EDSLs. A domain-specific language will typically have domain-specific errors about which the C++ compiler is ignorant[[40]](#footnote-40). Any errors the compiler is allowed to emit are likely to be too low-level to make any sense to the EDSL user (like that horrific Spirit error). The library needs to detect and report domain-specific errors. Fortunately, if you are using Boost.Proto, you have some powerful tools at your disposal[[41]](#footnote-41).[6](http://cpp-next.com/archive/2010/09/expressive-c-why-template-errors-suck-and-what-you-can-do-about-it/#fn:5) Let’s see in detail what my advice means for the EDSL I developed in the previous article.[[42]](#footnote-42)[[43]](#footnote-43)[[44]](#footnote-44)

**Compile-time vs. Runtime**

Why do good programmers ignore sound42 programming practice when they write templates?[5](http://cpp-next.com/archive/2010/09/expressive-c-why-template-errors-suck-and-what-you-can-do-about-it/#fn:4) From personal experience, I can say that it took a while before I appreciated43 the similarities between instantiating a template (compile-time computation) and calling a function (runtime computation), and even longer before I realized that engineering principles that apply at runtime—like parameter validation—can be applied at compile time, too. Now that I know it, it seems blindingly obvious.44

### Mad Libs[7](http://cpp-next.com/archive/2010/09/expressive-c-why-template-errors-suck-and-what-you-can-do-about-it/#fn:1) Formatting, Revisited

The [Mad Libs](http://www.madlibs.com/)-like string formatting API from the last article lets users format strings and specify map-like relationships inline. A typical usage looks like this:

std::cout << format("The home directory of {user} is {home}**\n**"

, map("user", "eric") ("home", "/home/eric") );

This expression should print:

The home directory of eric is /home/eric

The EDSL part is the second argument to format. Since we’ll be referring back to it a lot, I’ll duplicate the complete example from the last article:

|  |  |
| --- | --- |
| 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 49 50 | #include <map>  #include <string>  #include <iostream>  #include <boost/proto/proto.hpp>  #include <boost/xpressive/xpressive.hpp>  #include <boost/xpressive/regex\_actions.hpp>    struct map\_ {};  boost::proto::terminal<map\_>::type map = {};   typedef std::map<std::string, std::string> string\_map;   // Recursive function used to fill the map  template< class Expr >  void fill\_map( Expr const & expr, string\_map & subs )  {  using boost::proto::value; // read a value from a terminal  using boost::proto::child\_c; // get the Nth child of a non-terminal  subs[ value(child\_c<1>( expr )) ] = value(child\_c<2>(expr));  fill\_map( child\_c<0>(expr), subs );  }    // The 'map' terminal ends the recursion  void fill\_map( boost::proto::terminal<map\_>::type const &, string\_map & )  {}    // The old format API that accepts a map of string substitutions  std::string format( std::string fmt, string\_map & subs )  {  namespace xp = boost::xpressive;  using namespace xp;  sregex const rx = '{' >> (s1= +\_w) >> '}'; // like "{(\\w+)}"  return regex\_replace(fmt, rx, xp::ref(subs)[s1]); }    // The new format API that forwards to the old one  template< class Expr >  std::string format( std::string fmt, Expr const & expr )  {  string\_map subs;  fill\_map( expr, subs );  return format( fmt, subs );  }    int main()  {  std::cout << format("The home directory of {user} is {home}**\n**"  , map("user", "eric")  ("home", "/home/eric") );  } |

fill\_map expects to be given expression trees of a certain form. But notice how the second format overload takes the map expression and simply forwards it to fill\_map on line 41 without any parameter validation at all. Let’s mess[[45]](#footnote-45) with the expression tree and see what happens:

You might be thinking, “If you had simply defined the operator overload to only accept narrow strings, you would have gotten a better error message.” This is true. But with a simple technique discussed in later articles, you can restrict Proto’s operator overloads to only those that build trees that make sense in your domain. It will prove more flexible and powerful to do it that way.

std::cout << format("The home directory of {user} is {home}**\n**"

, map("user", L"eric")

("home", "/home/eric") );

Notice that I changed one string literal from narrow to wide. When I recompile the code with this most recent change, I get a 50+ line error message[8](http://cpp-next.com/archive/2010/09/expressive-c-why-template-errors-suck-and-what-you-can-do-about-it/#fn:6):



The error occurs deep within our EDSL implementation. Had we validated the expr parameter before calling fill\_map, we could have done much better. Let’s see how.

### Proto Grammars

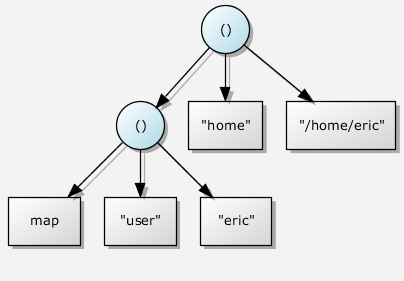
At first blush[[46]](#footnote-46), validating the expr parameter looks difficult. After all, the user could pass one of an infinite number of map expressions of arbitrary depth. But when we put our language-design goggles on[[47]](#footnote-47), this problem looks much simpler: we just need to find the grammar to which all map expressions must conform. Then we just check that the expression matches the grammar.

This expression:

map("user", "eric")

("home", "/home/eric")

…builds a Proto expression tree that looks like this:

[](http://cpp-next.com/wp-content/uploads/2010/09/03-fig1.dot_.notugly1.png)  
Figure 1: A map expression tree

In plain English, we can describe the structure of map expression trees as follows: a map expression is either:

* A map terminal, or
* A ternary function call where:
  + The 0th child is a valid map expression tree (note recursion),
  + The 1st child is a string, and
  + The 2nd child is also a string

Using Proto’s support for defining grammars, we can define the MapGrammar as follows (to be explained below):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | // Define the grammar of map expressions  struct MapGrammar  : proto::or\_<  // A map expression is a map terminal, or  proto::terminal<map\_>  // ... a ternary function non-terminal where the child nodes are  , proto::function<  // ... a map expression,  MapGrammar  // ... a narrow string terminal, and  , proto::terminal<char const \*>  // ... another narrow string terminal.  , proto::terminal<char const \*>  >  >  {}; |

Let’s take this one piece at a time:

* **Line 2:** Proto grammars are simple user-defined structs.
* **Line 3:** Inheritance is used to say that MapGrammar is expressed in terms of proto::or\_. Proto::or\_ is used for grammar alternation like the | operator in EBNF. An expression is allowed to match this or that. In Proto, alternate grammars are tried in order.
* **Line 5:** A map expression can be a simple map terminal. Note that proto::terminal<map\_> was used to define the global map object on line 9 of the [complete example](http://cpp-next.com/archive/2010/09/expressive-c-why-template-errors-suck-and-what-you-can-do-about-it/#example). It is also used here as a grammar that matches that terminal.
* **Line 7:** Proto::function defines a grammar that matches Proto expression nodes created by overloaded function-call operators. Proto provides templates like function for all the operators that Proto overloads. A complete list can be found in Proto’s [documentation](http://boost.org/doc/libs/1_44_0/doc/html/proto/users_guide.html#boost_proto.users_guide.intermediate_form.tags_and_metafunctions).
* **Line 9:** The zeroth child of the function node must match MapGrammar. This is interesting! It looks like we’re recursively defining MapGrammar in terms of itself. Surprisingly, this is legal. In fact, you may already be familiar with this technique. It’s called the [Curiously Recurring Template Pattern](http://en.wikipedia.org/wiki/Curiously_recurring_template_pattern), or CRTP. It gives Proto a natural way to define recursive grammars.
* **Lines 11-16:** Nothing too surprising here. The other two children must be narrow string terminals. The MapGrammar struct itself is empty. That is always the case for Proto grammars. [[48]](#footnote-48)[[49]](#footnote-49)[[50]](#footnote-50)[[51]](#footnote-51)[[52]](#footnote-52)[[53]](#footnote-53)[[54]](#footnote-54)[[55]](#footnote-55)[[56]](#footnote-56)

**Pause To Consider**

By now, you might be feeling a bit overwhelmed48. We just covered a lot of new ground49, and this coding style might feel strange to you. But consider for a moment what we’ve just expressed and how concisely we’ve expressed it: we’ve defined in code the grammar for valid map expressions, and it took only a pittance50 of code to do it. This is quite an accomplishment51. Take a moment to become comfortable with the definition of MapGrammar. Grammars are the central pillar52 of Proto, and once you get grammars under your belt53 (is that a pillar under your belt? har har54), you’ll really be able to do some neat55 things. In fact, all the powerful and interesting things you can do with Proto begin right here56 with grammars

### Validating Expressions Against Grammars

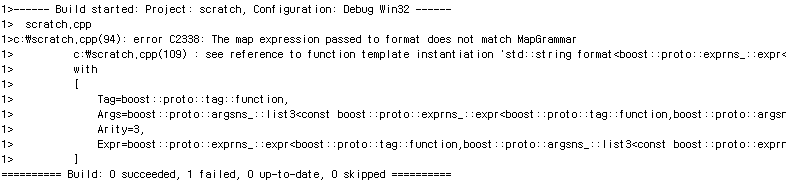
No doubt you’re wondering[[57]](#footnote-57) what we actually do with MapGrammar. Proto provides a trait called proto::matches for determining at compile time whether an expression type matches a given grammar. We can use proto::matches in conjunction with[[58]](#footnote-58) C++0x’s static\_assert or various C++03 approximations of it (see note below) to halt compilation as soon as an invalid expression is detected.

With static\_assert, proto::matches and MapGrammar, we can modify our format overload to validate the expr parameter before passing it to the fill\_map function:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | template< class Expr >  std::string format( std::string fmt, Expr const & expr )  {  */\* READ THIS IF YOUR COMPILE BREAKS ON THE FOLLOWING LINE*  *\**  *\* You have passed to format an invalid map expression.*  *\* They should be of the form:*  *\* map("this", "that")("here", "there")*  *\*/*  static\_assert(  proto::matches<Expr, MapGrammar>::value  , "The map expression passed to format does not match MapGrammar");  string\_map subs;  fill\_map( expr, subs );  return format( fmt, subs );  } |

When we pass our invalid expression to format now, our error goes from 50+ lines to about 10, including this message[9](http://cpp-next.com/archive/2010/09/expressive-c-why-template-errors-suck-and-what-you-can-do-about-it/#fn:7):

c:\scratch.cpp(94): error C2338: The map expression passed to format does not match MapGrammar



This error is nicer because:

* It is shorter!
* The error message itself indicates what the problem might be.
* The error happens at the API boundary, not on some random line of code deep in the library’s guts[[59]](#footnote-59).
* We’ve helpfully left a comment by the assert to let people know what’s wrong in case the assertion fails, and what to do to fix it.

If you don’t have a C++0x compiler with static\_assert, I recommend using [Boost.MPL](http://boost.org/libs/mpl)‘s [BOOST\_MPL\_ASSERT\_MSG](http://www.boost.org/doc/libs/1_44_0/libs/mpl/doc/refmanual/assert-msg.html) macro, which accepts a compile-time Boolean and a message to display if the Boolean is false. The static assertion on line 9 above would instead look like this:

BOOST\_MPL\_ASSERT\_MSG(

(proto::matches<Expr, MapGrammar>::value),

THE\_MAP\_EXPRESSION\_PASSED\_TO\_FORMAT\_DOES\_NOT\_MATCH\_MAPGRAMMAR,

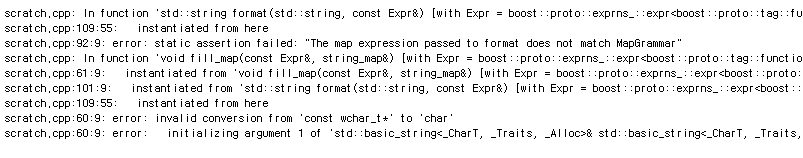
(MapGrammar));

When this assertion fails, it emits an error like:

c:\scratch.cpp(115): error C2664: 'boost::mpl::assertion\_failed' : cannot convert parameter 1 from 'boost::mpl::failed\*\*\*\*\*\*\*\*\*\*\*\*(\_\_thiscall format::THE\_MAP\_EXPRESSION\_PASSED\_TO\_FORMAT\_DOES\_NOT\_MATCH\_MAPGRAMMAR::\* \*\*\*\*\*\*\*\*\*\*\*)(MapGrammar)' to 'boost::mpl::assert::type'

### Avoid Follow-on Errors

If you try the above example on gcc-4.5, you’ll find that rather than a shorter error, the static\_assert gives a longer one!

What’s going on here? If you trawl[[60]](#footnote-60) through the error spew, you can see the nice message from the static assertion, but it’s buried in a lot of other junk with two other errors from the guts of our EDSL implementation. Let’s look again at the new implementation of format:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | template< class Expr >  std::string format( std::string fmt, Expr const & expr )  {  */\* READ THIS IF YOUR COMPILE BREAKS ON THE FOLLOWING LINE*  *\**  *\* You have passed to format an invalid map expression.*  *\* They should be of the form:*  *\* map("this", "that")("here", "there")*  *\*/*  static\_assert(  proto::matches<Expr, MapGrammar>::value  , "The map expression passed to format does not match MapGrammar");  string\_map subs;  fill\_map( expr, subs );  return format( fmt, subs );  } |

The static\_assert on line 10 causes the nice diagnostic, but gcc helpfully keeps right on compiling, eventually reaching the call to fill\_map on line 14. We’ve already established that the call will fail to compile, but nobody told gcc it was OK to stop!

In general, it’s not enough to issue a diagnostic for the known errors. We must also avoid the follow-on diagnostics from overeager compilers like gcc. The answer is usually quite simple: move the guts to a separate function, and use static dispatch to call that function or an empty one depending on whether validation succeeded. A little code should make it clear:

|  |  |
| --- | --- |
| 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 | template< class Expr >  std::string format\_impl( std::string fmt, Expr const & expr, boost::mpl::true\_ )  {  string\_map subs;  fill\_map( expr, subs );  return format( fmt, subs );  }    template< class Expr >  std::string format\_impl( std::string fmt, Expr const & expr, boost::mpl::false\_ )  {  return std::string(); // never called for valid imput  }    template< class Expr >  std::string format( std::string fmt, Expr const & expr )  {  */\* READ THIS IF YOUR COMPILE BREAKS ON THE FOLLOWING LINE*  *\**  *\* You have passed to format an invalid map expression.*  *\* They should be of the form:*  *\* map("this", "that")("here", "there")*  *\*/*  static\_assert(  proto::matches<Expr, MapGrammar>::value  , "The map expression passed to format does not match MapGrammar");    */\* Dispatch to the real implementation or a stub depending on*  *whether our parameters are valid or not.*  *\*/*  return format\_impl( fmt, expr, proto::matches<Expr, MapGrammar>() );  } |

We added two overloads of a new function format\_impl. The first takes an extra argument of type boost::mpl::true\_ and does the real work. The second takes boost::mpl::false\_ and simply returns an empty string. The original format function is now just a shell that (maybe) issues a diagnostic and dispatches to one or the other overload. Proto::matches conveniently inherits from mpl::true\_ or mpl::false\_ accordingly to make this possible. With this change, the full error is much shorter:[[61]](#footnote-61)[[62]](#footnote-62)[[63]](#footnote-63)

It sucks that we have to muddy61 our code to accommodate62 gcc like this, but it’s a small price. We’ve already done the hard work of validating the parameters. Avoiding follow-on errors is always a simple matter of dispatching on the compile-time information we have already calculated anyway. And remember your users; they deserve63 nice error messages.

scratch.cpp: In function ‘std::string format(std::string, const Expr&) [with = ...

scratch.cpp:126:55: instantiated from here

scratch.cpp:112:9: error: static assertion failed: "The map expression passed to format does not match MapGrammar"

### Conclusions and What's To Come

Thanks for reading. Since I'm priming[[64]](#footnote-64) you guys to be library authors, I feel obligated[[65]](#footnote-65) to give you the tools to make your libraries user-friendly. As you can see, we had to be a bit proactive[[66]](#footnote-66) about making our code behave well when passed garbage, but it wasn't so hard. Although I talked mostly about EDSLs and Proto, these techniques are applicable very broadly:

1. Validate template parameters at API boundaries.
2. Use C++0x's static\_assert or a C++03 equivalent to issue readable diagnostics.
3. Leave detailed comments by the static assertions to let people know what has gone wrong and how to fix it.
4. Dispatch to stubs on invalid input to avoid follow-on failures.

These techniques can greatly reduce the amount of compiler spew C++ programmers encounter on a daily basis.

Proto grammars make validating expression trees easy and (dare I say it?[[67]](#footnote-67)) fun. But they are far more useful than that. You can use Proto grammars to restrict Proto's operator overloads to only those that create valid trees. And by embedding semantic actions (a.k.a transforms) within Proto grammars, you can write algorithms that manipulate trees and generate code in powerful ways. In future articles, we'll dig deep into Proto grammars and transforms. But first, we'll take a closer look at Proto expressions and how to extend them, adding and customizing member functions, making them anything but dumb, static trees[[68]](#footnote-68).

Until next time, don't forget to validate your parameters. And if you see any bad template errors, file a bug!

1. Previous articles in this series referred to them as domain-specific embedded languages, or DSELs. Due to feedback I've received, I've decided to switch to the more commonly-used term EDSL. I'm still referring the same thing. [↩](http://cpp-next.com/archive/2010/09/expressive-c-why-template-errors-suck-and-what-you-can-do-about-it/#fnref:8)
2. A knowing reference to Neal Stephenson's fictional [Metaverse](http://en.wikipedia.org/wiki/Metaverse) in his book "Snow Crash", one of my personal faves. [↩](http://cpp-next.com/archive/2010/09/expressive-c-why-template-errors-suck-and-what-you-can-do-about-it/#fnref:2)
3. [Boost.StaticAssert](http://www.boost.org/doc/libs/1_44_0/doc/html/boost_staticassert.html) and [Boost.Concept\_check](http://www.boost.org/doc/libs/1_44_0/libs/concept_check/concept_check.htm), for example. [↩](http://cpp-next.com/archive/2010/09/expressive-c-why-template-errors-suck-and-what-you-can-do-about-it/#fnref:10)
4. Compiled with g++ 4.5.0 against Boost trunk as of Sept 16, 2010. [↩](http://cpp-next.com/archive/2010/09/expressive-c-why-template-errors-suck-and-what-you-can-do-about-it/#fnref:3)
5. I'm sorry to pick on the Spirit authors here, but I had to pick on somebody. Spirit is actually better than most at catching and reporting invalid input, and finding this particular chink[[69]](#footnote-69) in Spirit's armor[[70]](#footnote-70) took some fiddling[[71]](#footnote-71). [↩](http://cpp-next.com/archive/2010/09/expressive-c-why-template-errors-suck-and-what-you-can-do-about-it/#fnref:4)
6. I'm not claiming Boost.Proto is a silver bullet. It provides the tools. You have to use them. Boost.Spirit is actually implemented with Proto, but that didn't help in this example. It appears that Spirit is letting this invalid input through and failing badly in deeply nested template instantiations. [↩](http://cpp-next.com/archive/2010/09/expressive-c-why-template-errors-suck-and-what-you-can-do-about-it/#fnref:5)
7. Mad Libs is a registered trademark of Penguin Group (USA) Inc. [↩](http://cpp-next.com/archive/2010/09/expressive-c-why-template-errors-suck-and-what-you-can-do-about-it/#fnref:1)
8. Tested with Microsoft Visual C++ 2010 [↩](http://cpp-next.com/archive/2010/09/expressive-c-why-template-errors-suck-and-what-you-can-do-about-it/#fnref:6)
9. Tested with Microsoft Visual C++ 2010 [↩](http://cpp-next.com/archive/2010/09/expressive-c-why-template-errors-suck-and-what-you-can-do-about-it/#fnref:7)

1. suck : (속어) 엉망이다, 형편없다. [↑](#footnote-ref-1)
2. then = EDSL [↑](#footnote-ref-2)
3. ~ is intentionally provocative : ~ 은 의도된 도발이다 [↑](#footnote-ref-3)
4. creative license : 장치 In writing, use or making of interesting ideas [↑](#footnote-ref-4)
5. righteous rage : 당연한 분노 [↑](#footnote-ref-5)
6. lay some much-deserved blame : 비난 받아 마땅한 몇 가지를 늘어 놓다 [↑](#footnote-ref-6)
7. catharsis : 카타르시스. 아리스토텔레스의 ‘시학’에 나타난 비극 이론의 한 중요한 개념. 청중은 비극에서 고통의 장면을 보게 됨으로 인해 흥분하거나 우울해 하지 않고, 그 반대로 일종의 해방감을 느끼게 됨. 지적으로 잘 조절된 행위의 구성에 의해 우리의 감정이 허락되고 해방된다는 사실에서 비극 고유의 쾌락이 발생됨. [↑](#footnote-ref-7)
8. ire : 분노 [↑](#footnote-ref-8)
9. state of affairs : 사태, 사정 [↑](#footnote-ref-9)
10. meta-verse =cyberspace [↑](#footnote-ref-10)
11. yawn **:** 하품 나는[따분한] 일[것] [↑](#footnote-ref-11)
12. spew : 거침없이 토해 낸 것. 토사물 [↑](#footnote-ref-12)
13. take your pick : 골라 봐라. [↑](#footnote-ref-13)
14. their share of the heat : 그 압박감에 대한 그들의 몫 [↑](#footnote-ref-14)
15. The key selling point : 핵심 장점 [↑](#footnote-ref-15)
16. R.I.P. : 고이 잠드소서. R.I.P. is written on gravestones and expresses the hope that the person buried there may rest in peace. R.I.P. is an abbreviation for the Latin expression `requiescat in pace' or `requiescant in pace'. [↑](#footnote-ref-16)
17. begins at home : 동네에서 시작되다. [↑](#footnote-ref-17)
18. for a while : 얼마 동안 [↑](#footnote-ref-18)
19. folks : 사람들 [↑](#footnote-ref-19)
20. settle for : 불만스럽지만) 받아들이다. (꼭 원하는 것은 아니지만) ~에 만족하다 [↑](#footnote-ref-20)
21. we’d be outraged : 화가 날 만 하다(?) [↑](#footnote-ref-21)
22. in a way : 어떤 면에서는 [↑](#footnote-ref-22)
23. that’s likely to raise a few eyebrows : 다소 눈살을 찌푸릴 지도 모른다. [↑](#footnote-ref-23)
24. curse : 악담하다. [↑](#footnote-ref-24)
25. file bugs : 버그를 보고하다. [↑](#footnote-ref-25)
26. hey : 보자 [↑](#footnote-ref-26)
27. endemic : (해당 영역의)고질병 [↑](#footnote-ref-27)
28. simply put : 간단히 말해서 [↑](#footnote-ref-28)
29. offending : 불쾌하게 하는, 문제가 되는. [↑](#footnote-ref-29)
30. remedial : 개선을 위한 [↑](#footnote-ref-30)
31. 101 : 기초. The term **101** (pronounced "One-oh-one") often indicates an introductory level of learning. [↑](#footnote-ref-31)
32. exploit : 부당하게 이용하다. [↑](#footnote-ref-32)
33. worth his salt will tell you this : ? [↑](#footnote-ref-33)
34. blithely : 기꺼이. 즐거이. 쾌활히. [↑](#footnote-ref-34)
35. car wreck of epic proportions : 엄청난 규모의 자동차 사고 [↑](#footnote-ref-35)
36. spot the typo : 오타를 찾다. [↑](#footnote-ref-36)
37. Answer : [It’s the tilde before factor on line 9.] [↑](#footnote-ref-37)
38. screaming : 날카롭게 외치는 [↑](#footnote-ref-38)
39. acute : 잘 발달된. [↑](#footnote-ref-39)
40. ignorant : 알지 못하는 [↑](#footnote-ref-40)
41. at your disposal : 원하는 대로 쓸 수 있는 [↑](#footnote-ref-41)
42. sound : 유의미한. [ADJ] ****advice, reasoning, or evidence is reliable and sensible. [↑](#footnote-ref-42)
43. appreciate : 진가를 알아보다. [↑](#footnote-ref-43)
44. blindingly obvious : 아주 명백하다. [↑](#footnote-ref-44)
45. mess : 엉망으로 만들다. [↑](#footnote-ref-45)
46. at first blush : 언뜻 보기에 [↑](#footnote-ref-46)
47. put goggles on : 고글을 끼다. [↑](#footnote-ref-47)
48. overwhelmed : 압도된 [↑](#footnote-ref-48)
49. new-ground : 신개지, 신개척(개간)지. [↑](#footnote-ref-49)
50. pittance : 아주 적은 [↑](#footnote-ref-50)
51. quite an accomplishment : 대단한 업적 [↑](#footnote-ref-51)
52. central pillar : 중심 축 [↑](#footnote-ref-52)
53. under your belt : 겸비하다 [↑](#footnote-ref-53)
54. har har : 와하하((큰웃음)). [↑](#footnote-ref-54)
55. neat : 세련된. [↑](#footnote-ref-55)
56. right here: 지금 당장 [↑](#footnote-ref-56)
57. No doubt you’re wondering : 틀림 없이 궁금해 하고 있을 것이다. [↑](#footnote-ref-57)
58. in conjunction with ~ : ~와 함께. [↑](#footnote-ref-58)
59. gut : 내장. 소화기관. [↑](#footnote-ref-59)
60. trawl : 샅샅이 훑다. [↑](#footnote-ref-60)
61. muddy : 오염시키다. [↑](#footnote-ref-61)
62. accommodate : (의견 등을) 수용하다[담다]. (환경 등에) 맞추다 [↑](#footnote-ref-62)
63. deserve : 누릴 자격이 있다. [↑](#footnote-ref-63)
64. be priming : 준비시키다. [↑](#footnote-ref-64)
65. obligated : 의무가 있는. [↑](#footnote-ref-65)
66. proactive : 미리 적극적으로. [↑](#footnote-ref-66)
67. dare I say it? : 아마 이렇게 말해도 되겠죠? [↑](#footnote-ref-67)
68. anything but dumb, static trees : 결코 쓸모가 없지 않은 정적 트리. [↑](#footnote-ref-68)
69. chink : 균열 [↑](#footnote-ref-69)
70. armor : 갑옷 [↑](#footnote-ref-70)
71. fiddling : 하찮은, 시시한. [↑](#footnote-ref-71)