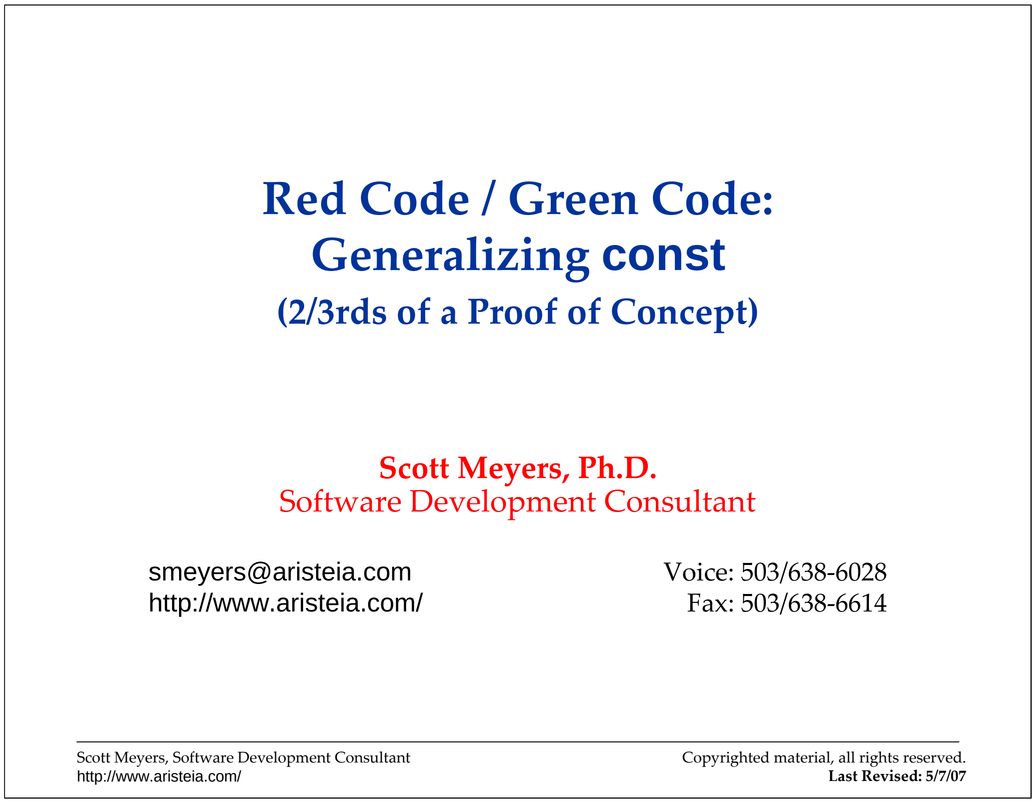


Red Code / Green Code: Generalizing const

(2/3rds of a Proof of Concept)

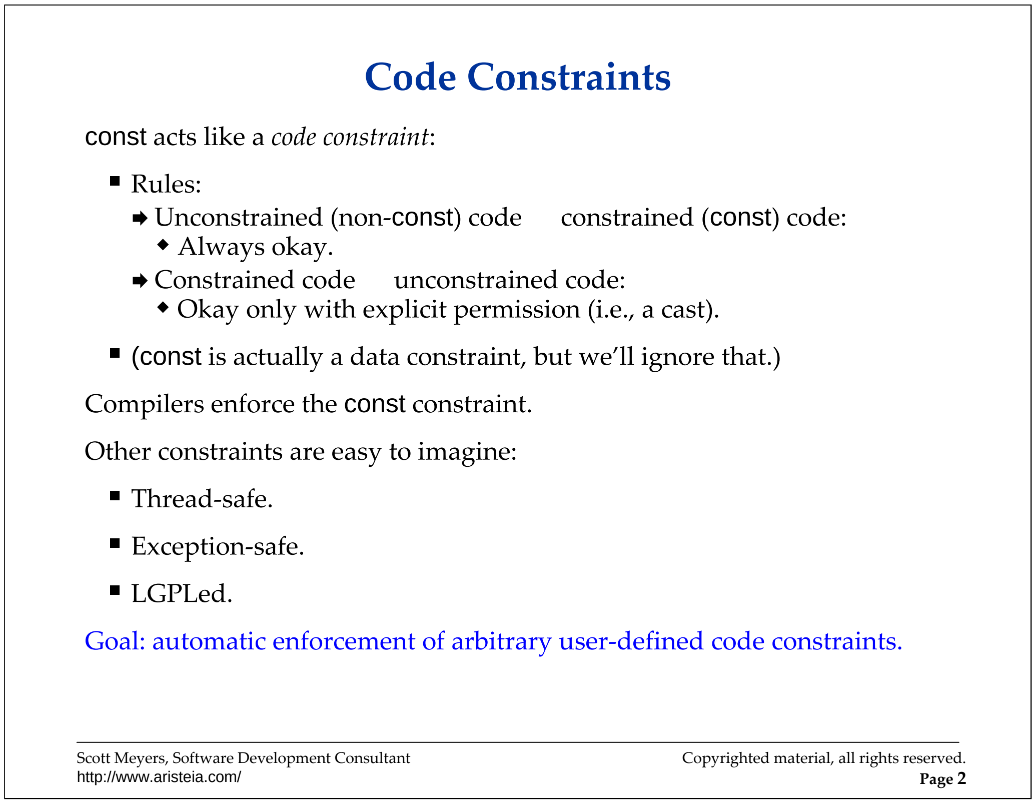
Scott Meyers, Ph.D.
Software Development Consultant

smeyers@aristeia.com
http://www.aristeia.com/

Voice: 503/638-6028
Fax: 503/638-6614

After the talk, Herb Sutter showed me a solution to this problem that I like better than the one I presented. Herb describes his approach in a blog entry at http://herbsutter.spaces.live.com/blog/cns!2D4327CC297151BB!207.entry. I summarize his approach later in these notes (on the slides where it's germane). I have not tested Herb's solution, but I'm confident that it (or something quite similar to it) will work

I've removed the film clip slides from this version of the presentation, because they don't add anything to a printed treatment.



Code Constraints

const acts like a *code constraint*:

- Rules:
  - Unconstrained (non-const) code   constrained (const) code:
    - Always okay.
  - Constrained code   unconstrained code:
    - Okay only with explicit permission (i.e., a cast).
- (const is actually a data constraint, but we'll ignore that.)

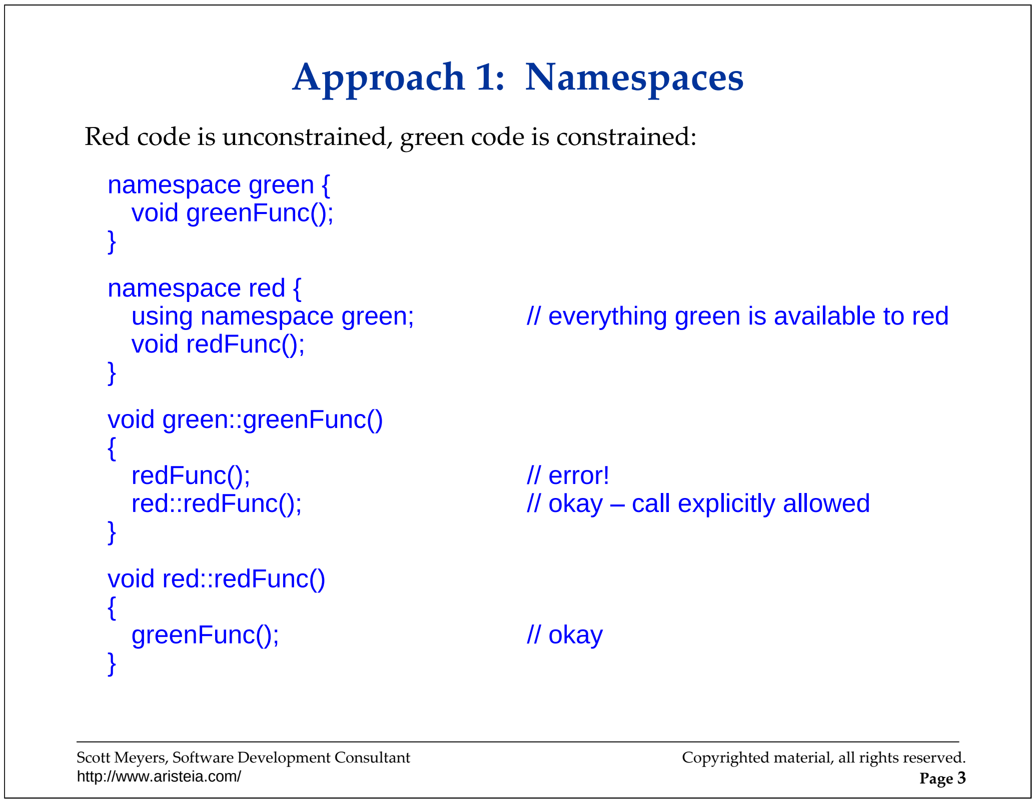
Compilers enforce the const constraint.

Other constraints are easy to imagine:

- Thread-safe.
- Exception-safe.
- LGPLed.

Goal: automatic enforcement of arbitrary user-defined code constraints.

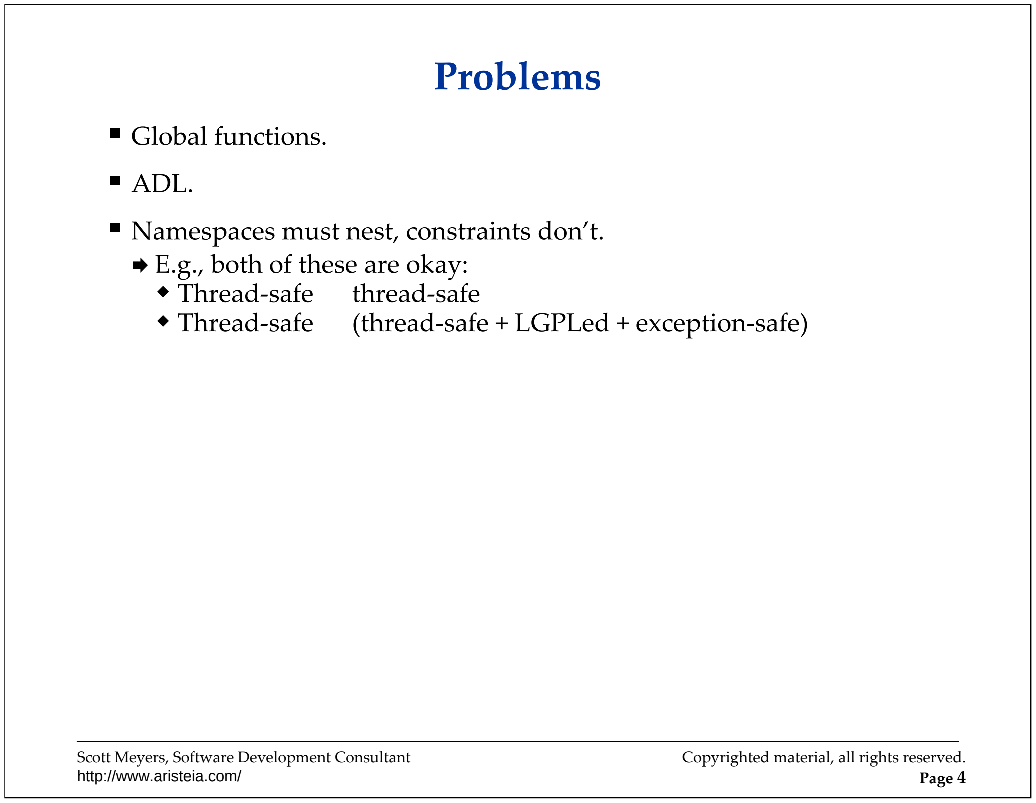
LGPLed code is covered by the LGPL. Such code can be called by non-GPLed code, but my understanding is that if LGPLed code calls non-LGPLed code, the called code must be made available under the GPL.



Approach 1: Namespaces

Red code is unconstrained, green code is constrained:

```
namespace green {
   void greenFunc();
}

namespace red {
   using namespace green;           // everything green is available to red
   void redFunc();
}

void green::greenFunc()
{
   redFunc();                       // error!
   red::redFunc();                  // okay – call explicitly allowed
}

void red::redFunc()
{
   greenFunc();                     // okay
}
```

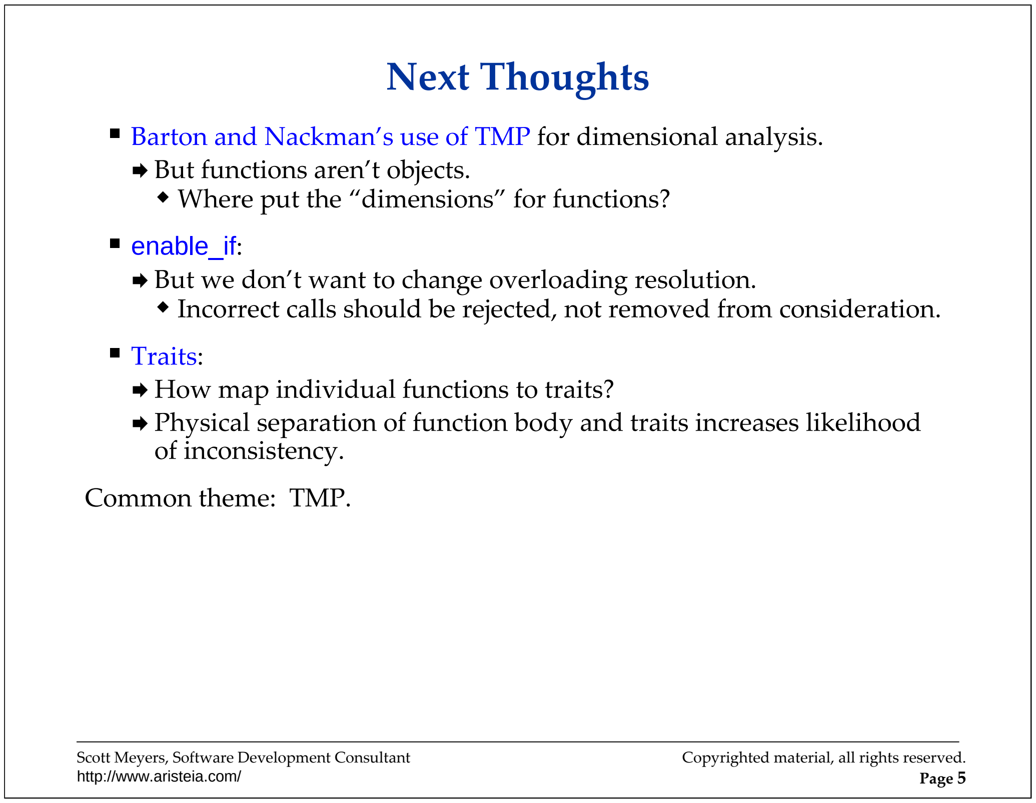


Problems

- Global functions.
- ADL.
- Namespaces must nest, constraints don't.
  - E.g., both of these are okay:
    - Thread-safe   thread-safe
    - Thread-safe   (thread-safe + LGPLed + exception-safe)

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 4



Next Thoughts

- Barton and Nackman's use of TMP for dimensional analysis.
  - But functions aren't objects.
    - Where put the "dimensions" for functions?
- enable_if:
  - But we don't want to change overloading resolution.
    - Incorrect calls should be rejected, not removed from consideration.
- Traits:
  - How map individual functions to traits?
  - Physical separation of function body and traits increases likelihood of inconsistency.

Common theme: TMP.

Approach 2: Types as Constraints

Basic idea:

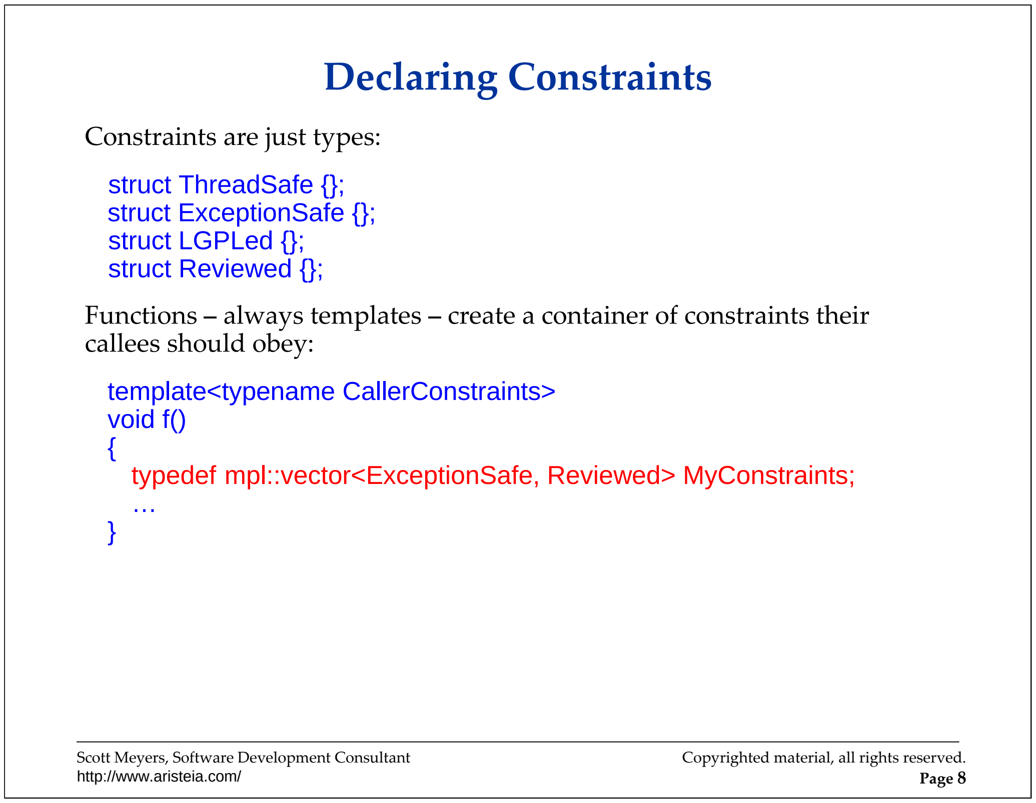
- Constraints are represented by UDTs.
  - ➡ Just like STL iterator categories.
- A function’s constraints is represented by a set of types.
- Caller   callee is okay iff (caller constraints)   (callee constraints).
  - ➡ Use TMP to enforce this during compilation.

The MPL

Boost's MPL offers STL-like functionality for TMP:

- Containers of types (e.g., mpl::vector, mpl::list, etc.)
- Algorithms over containers (e.g., mpl::find, mpl::equal, etc.)
- Iterators over containers.
- Etc.

Using the MPL definitely easier than "bare metal" TMP.



Declaring Constraints

Constraints are just types:

```
struct ThreadSafe {};
struct ExceptionSafe {};
struct LGPLed {};
struct Reviewed {};
```

Functions – always templates – create a container of constraints their callees should obey:

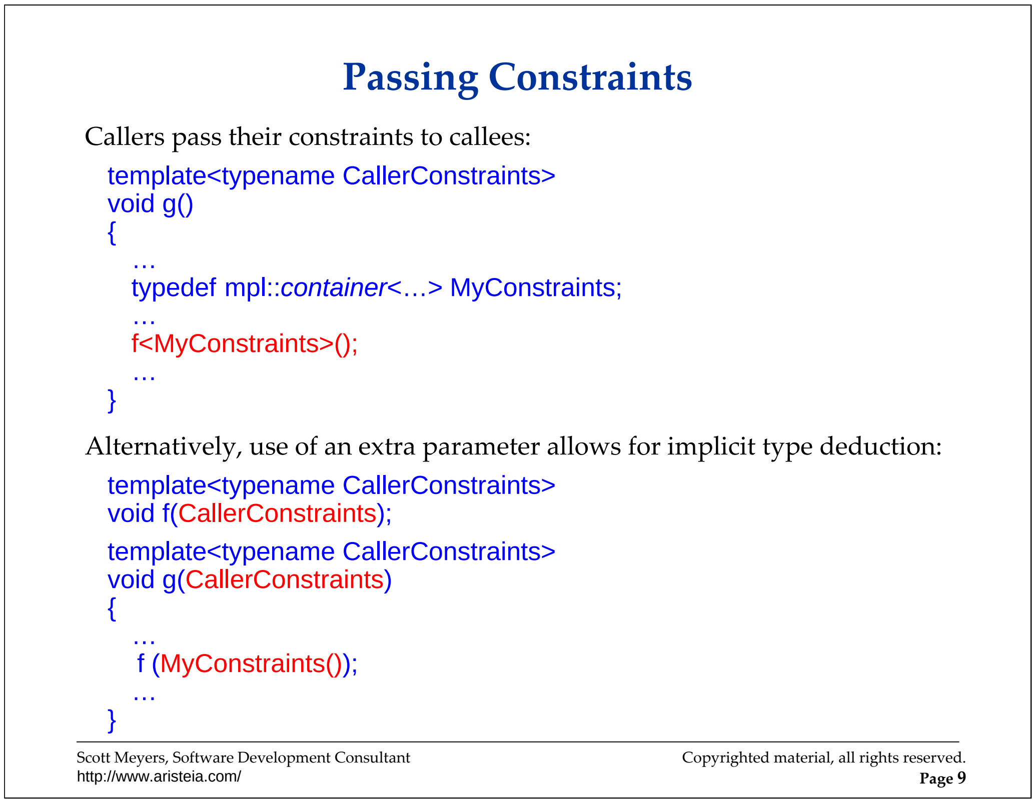
```
template<typename CallerConstraints>
void f()
{
  typedef mpl::vector<ExceptionSafe, Reviewed> MyConstraints;
  …
}
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/
Copyrighted material, all rights reserved.
Page 8

Herb's solution is based on functions declaring an additional parameter of a type that includes the collection of constraints. For example:

```
template<typename Constraints>      // details provided later
struct CallConstraints{
    …
};

void f(CallConstraints<mpl::vector<ExceptionSafe, Reviewed> >);
```

This approach addresses two concerns that came up during the talk. First, f is no longer a template, so template-induced code bloat is no longer an issue. Second, f's constraints are part of its declaration rather than being buried in its definition.



Passing Constraints

Callers pass their constraints to callees:

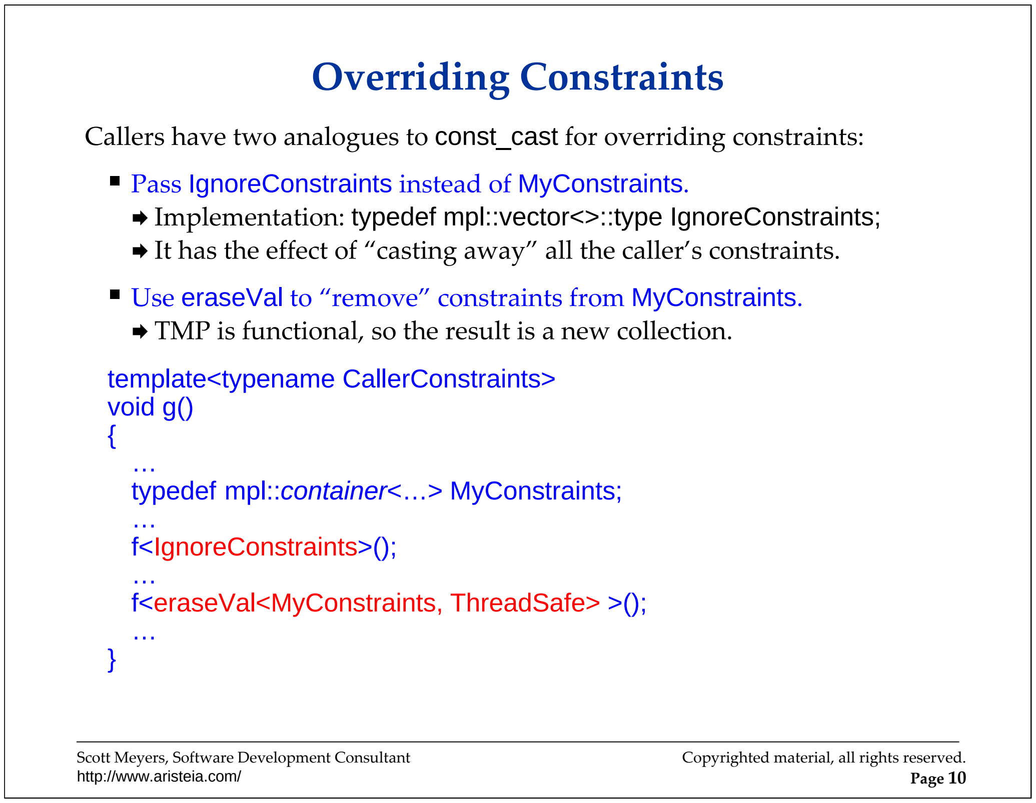
```
template<typename CallerConstraints>
void g()
{
   …
   typedef mpl::container<…> MyConstraints;
   …
   f<MyConstraints>();
   …
}
```

Alternatively, use of an extra parameter allows for implicit type deduction:

```
template<typename CallerConstraints>
void f(CallerConstraints);

template<typename CallerConstraints>
void g(CallerConstraints)
{
   …
    f (MyConstraints());
   …
}
```

With Herb's design, callers use a syntax similar to the second one above. The call from g to f, for example, would be written like this:

```
f(CallConstraints<MyConstraints>());
```



Overriding Constraints

Callers have two analogues to const_cast for overriding constraints:

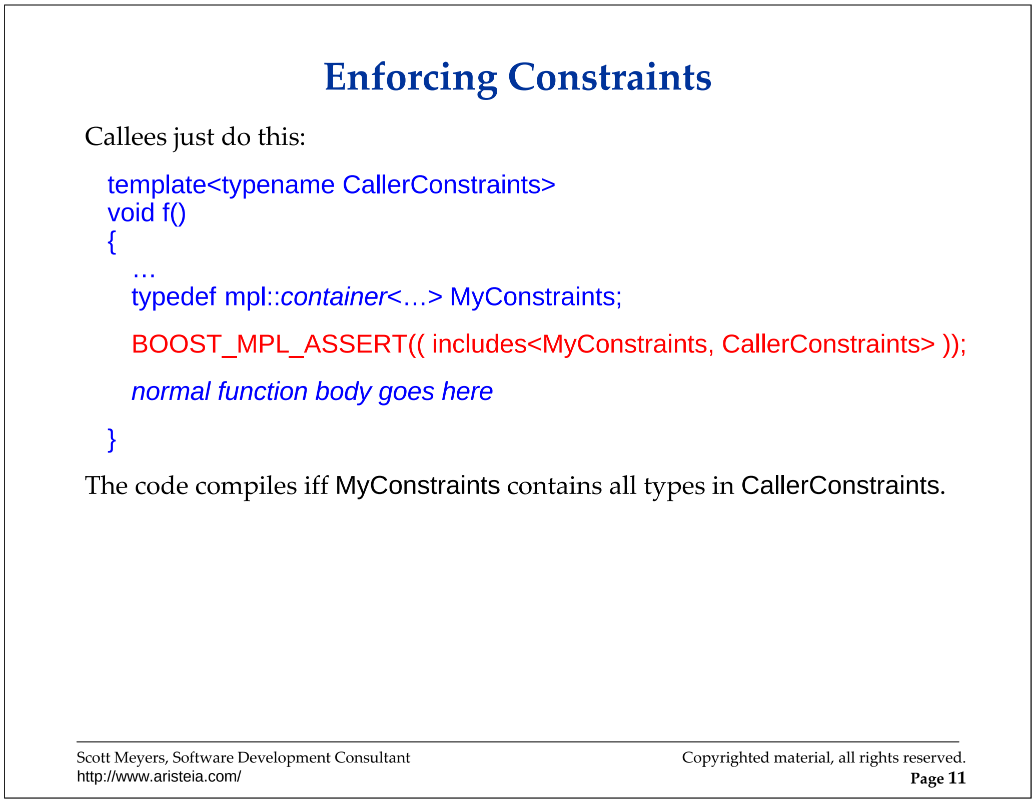
- Pass IgnoreConstraints instead of MyConstraints.
  - Implementation: typedef mpl::vector<>::type IgnoreConstraints;
  - It has the effect of "casting away" all the caller's constraints.
- Use eraseVal to "remove" constraints from MyConstraints.
  - TMP is functional, so the result is a new collection.

```
template<typename CallerConstraints>
void g()
{
  …
  typedef mpl::container<…> MyConstraints;
  …
  f<IgnoreConstraints>();
  …
  f<eraseVal<MyConstraints, ThreadSafe> >();
  …
}
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 10

The above ideas work equally well with Herb's design.



Enforcing Constraints

Callees just do this:

```
template<typename CallerConstraints>
void f()
{
  …
  typedef mpl::container<…> MyConstraints;

  BOOST_MPL_ASSERT(( includes<MyConstraints, CallerConstraints> ));

  normal function body goes here

}
```

The code compiles iff MyConstraints contains all types in CallerConstraints.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 11

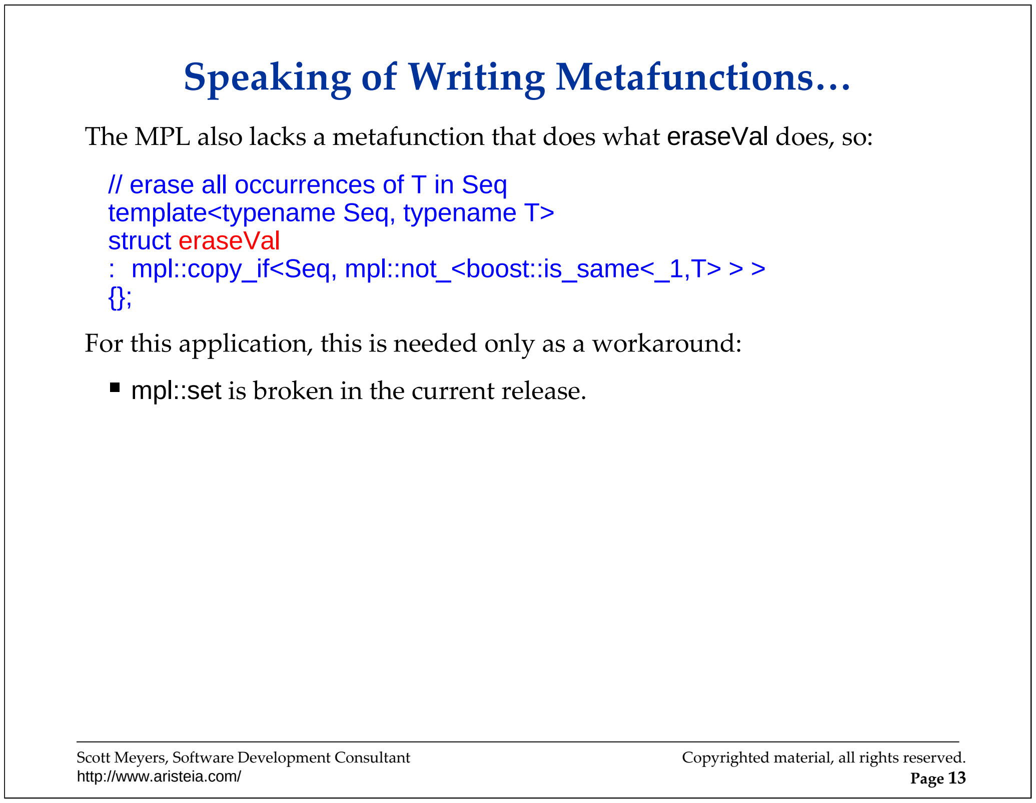
Constraint enforcement in Herb's design is based on implicit type conversion. We design things so that a call compiles iff there is an implicit conversion from CallConstraints<CallerConstraints> to CallConstraints<CalleeConstraints>. Like this:

```
template<typename Constraints>
struct CallConstraints  {

    template<typename CalleeConstraints>
    operator CallConstraints<CalleeConstraints>() const
    {
        BOOST_MPL_ASSERT((includes<CalleeConstraints,Constraints>));

        return CallContraints<CalleeConstraints>();

    }

};
```

Enforcing Constraints

The MPL has no includes metafunction, so:

```
// returns whether Sup includes each of the elements in Sub
template<typename Sup,                  // the putative superset
         typename Sub>                  // the putative subset
struct includes
:  boost::is_same<
     typename mpl::find_if<Sub,
                           mpl::not_<mpl::contains<Sup, _1> > >::type,

     typename mpl::end<Sub>::type
   >
{};
```



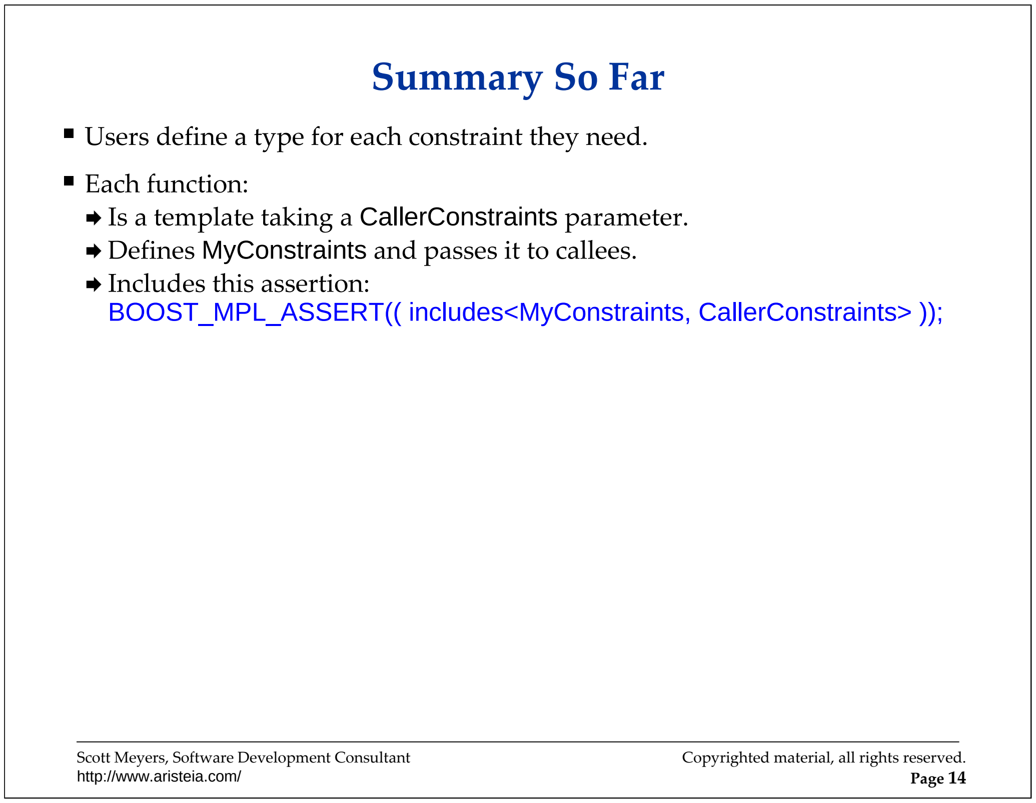
Speaking of Writing Metafunctions…

The MPL also lacks a metafunction that does what eraseVal does, so:

```
// erase all occurrences of T in Seq
template<typename Seq, typename T>
struct eraseVal
:  mpl::copy_if<Seq, mpl::not_<boost::is_same<_1,T> > >
{};
```

For this application, this is needed only as a workaround:

- mpl::set is broken in the current release.



Summary So Far

- Users define a type for each constraint they need.
- Each function:
  - ➨ Is a template taking a CallerConstraints parameter.
  - ➨ Defines MyConstraints and passes it to callees.
  - ➨ Includes this assertion:

```
BOOST_MPL_ASSERT(( includes<MyConstraints, CallerConstraints> ));
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

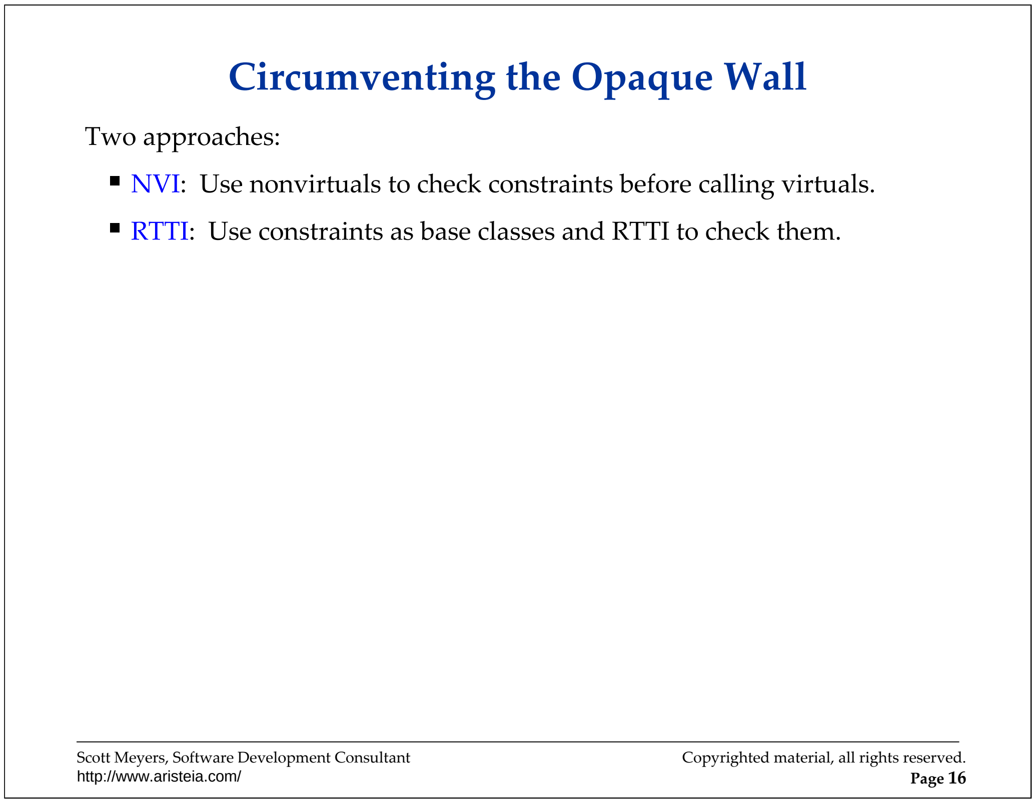
Copyrighted material, all rights reserved.
Page 14

Using Herb's design, users still define a type for each constraint they need, but functions need simply declare an extra parameter of type CallConstraints<MyConstraints>. Everything else is handled automatically.

Dealing with Virtual Functions

From the Boost.Spirit doc:

- *The virtual function is the meta-programmer's hell. … it is … an opaque wall where no metaprogram can get past. It kills all meta-information beyond the virtual function call.*

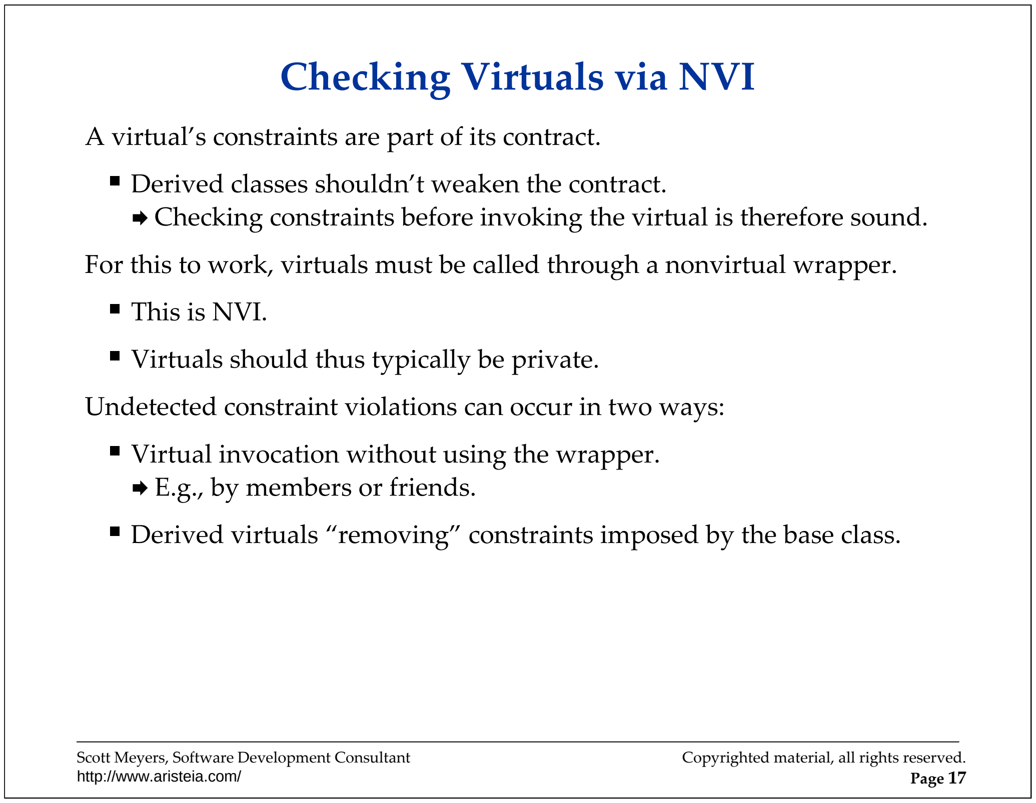


Circumventing the Opaque Wall

Two approaches:

- NVI: Use nonvirtuals to check constraints before calling virtuals.
- RTTI: Use constraints as base classes and RTTI to check them.

Because Herb's design includes the constraints in the function's parameter list, virtual functions don't need any special treatment: virtual overrides in derived classes have to declare the same constraints as the original virtual function. The drawback to this approach is that if derived virtuals want to add new constraints, they can't add them to their signatures. They can, however, continue to define a local MyConstraints typedef that includes the additional constraints and use that when making calls to other functions.



Checking Virtuals via NVI

A virtual's constraints are part of its contract.

- Derived classes shouldn't weaken the contract.
  - ➡ Checking constraints before invoking the virtual is therefore sound.

For this to work, virtuals must be called through a nonvirtual wrapper.

- This is NVI.
- Virtuals should thus typically be private.

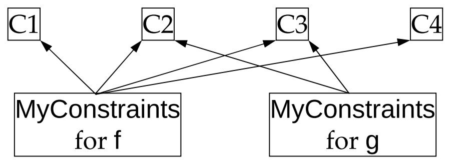
Undetected constraint violations can occur in two ways:

- Virtual invocation without using the wrapper.
  - ➡ E.g., by members or friends.
- Derived virtuals "removing" constraints imposed by the base class.

Checking Virtuals via RTTI

My initial idea:

- Caller and callee create objects that inherit from all their constraints.
  - ➡ Inspiration: Acyclic Visitor.



g should be able to call f, but not vice-versa.

- Constraints are satisfied iff callee object has all bases caller object does.

Problem:

- How perform the test?
  - ➡ Constraint classes are arbitrary UDTs.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/
Copyrighted material, all rights reserved.
Page 18

Checking Virtuals via TMP+RTTI

My next idea:

- Use TMP to generate the appropriate dynamic_casts.

Challenge:

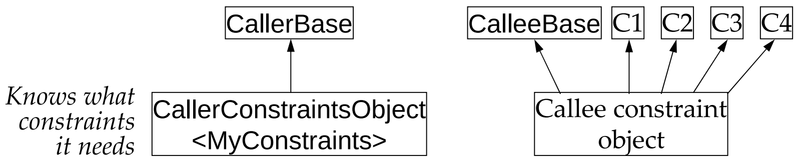
- Only caller knows what base classes to dynamic_cast to.
- No way for caller to get callee constraints.
- No way for callee to get caller constraints during compilation.
  - ➡ (For virtual functions.)

Checking Virtuals via TMP+RTTI

Approach:

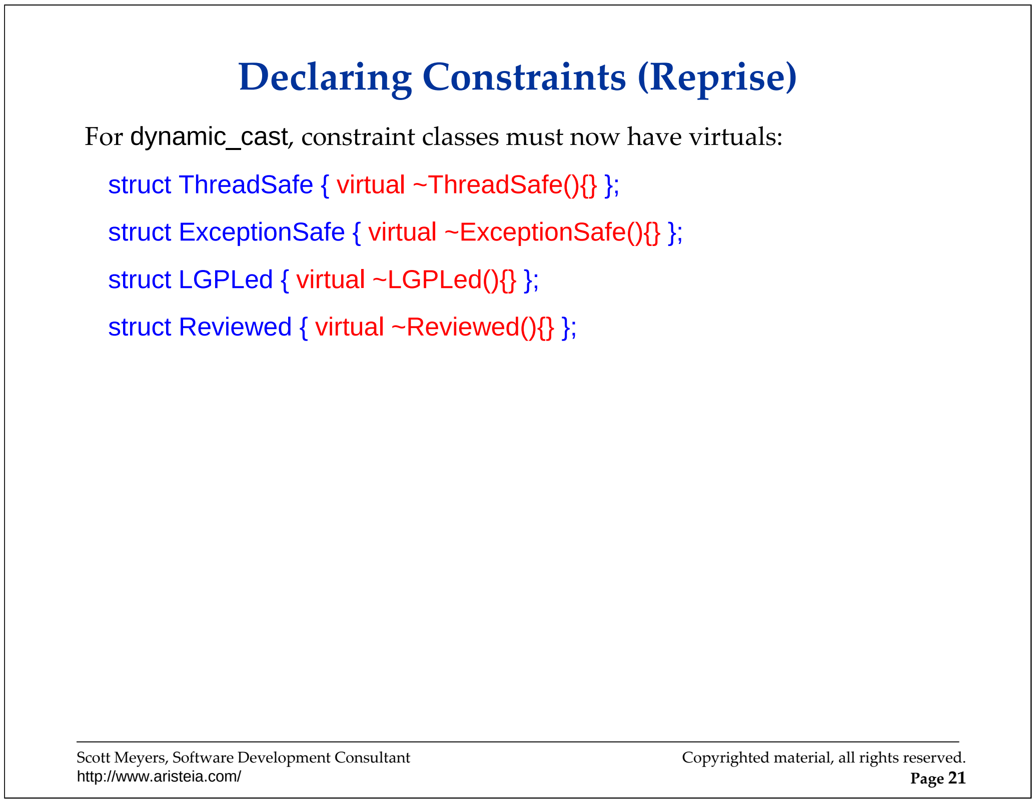
- Caller uses TMP to generate dynamic_casts inside its constraint object.
- Caller passes its constraint object to callee.
- Callee has caller object perform constraint checking.

Conceptually:


CallerBase
Knows what constraints it needs
CallerConstraintsObject <MyConstraints>
CalleeBase
C1
C2
C3
C4
Callee constraint object

Inspiration:

- Implementation of Boost's shared_ptr.



Declaring Constraints (Reprise)

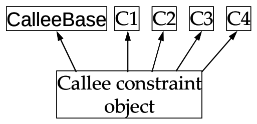
For dynamic_cast, constraint classes must now have virtuals:

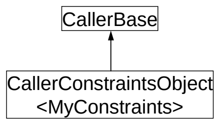
```
struct ThreadSafe { virtual ~ThreadSafe(){} };
struct ExceptionSafe { virtual ~ExceptionSafe(){} };
struct LGPLed { virtual ~LGPLed(){} };
struct Reviewed { virtual ~Reviewed(){} };
```

Caller and Callee Constraint Object Base Classes

CalleeBase need support only dynamic_cast:

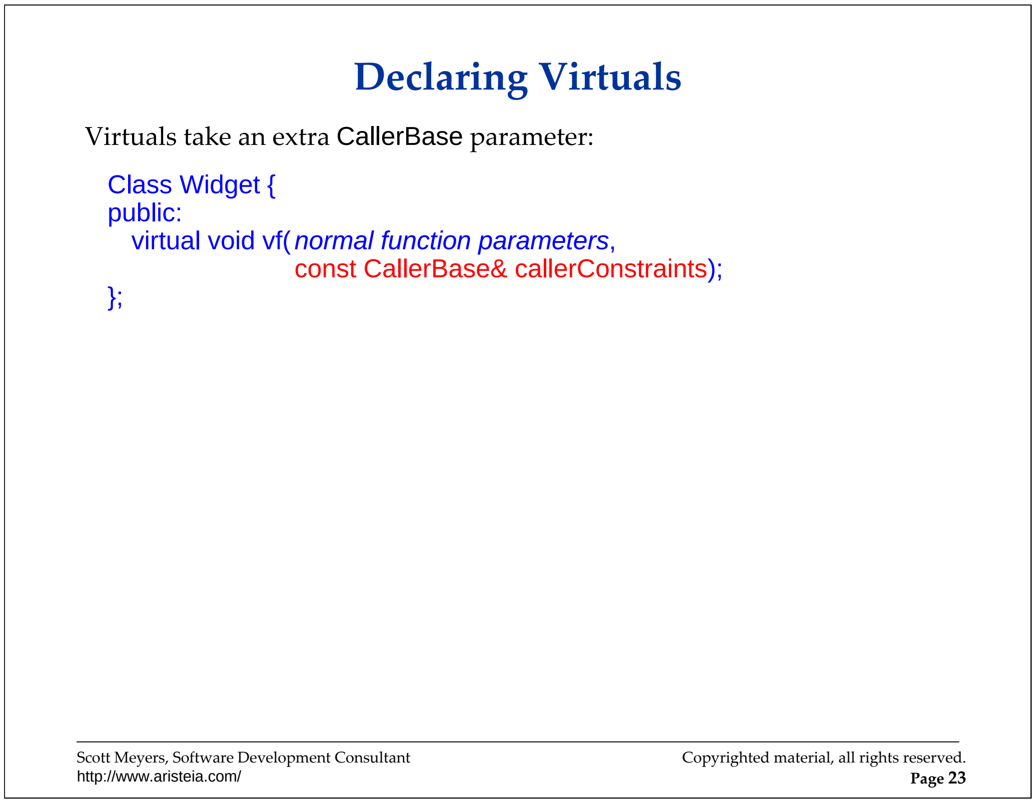
```
struct CalleeBase {
  virtual ~CalleeBase(){}
};
```





CallerBase supports constraint checking:

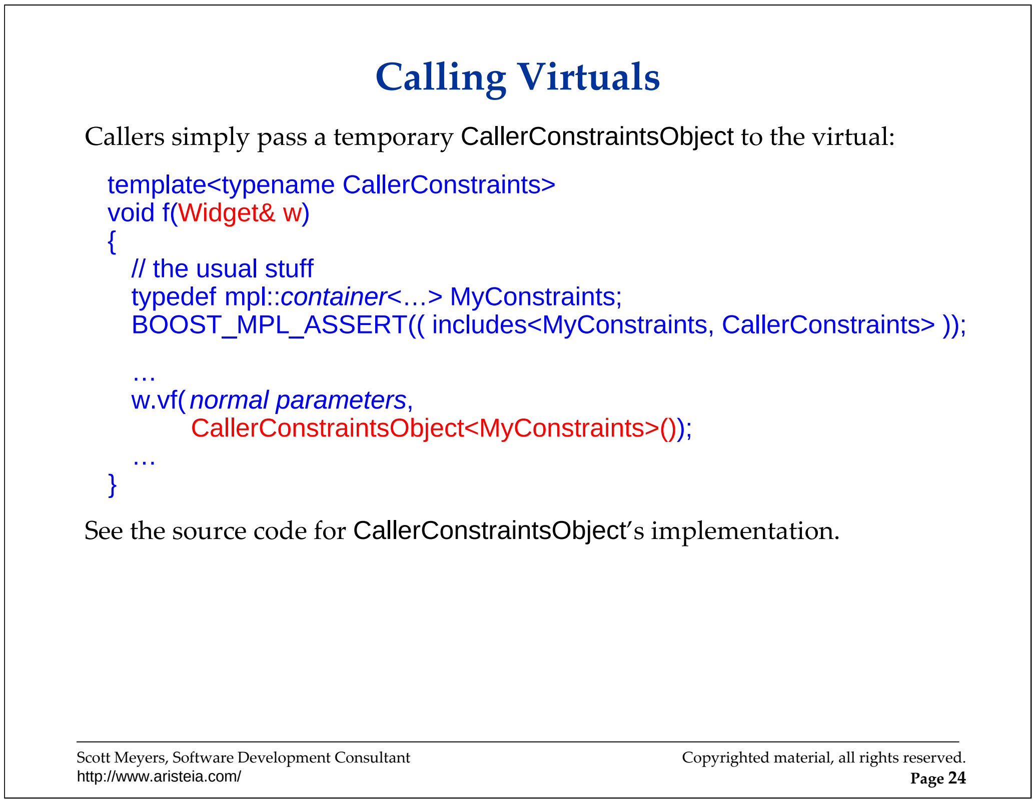
```
struct CallerBase {

  // return whether calleeTypesObj inherits from all constraint classes.
  virtual bool
  areIncludedBy(const CalleeBase& calleeTypesObj) const = 0;

};
```



Declaring Virtuals

Virtuals take an extra CallerBase parameter:

```
Class Widget {
public:
  virtual void vf(normal function parameters,
                  const CallerBase& callerConstraints);
};
```

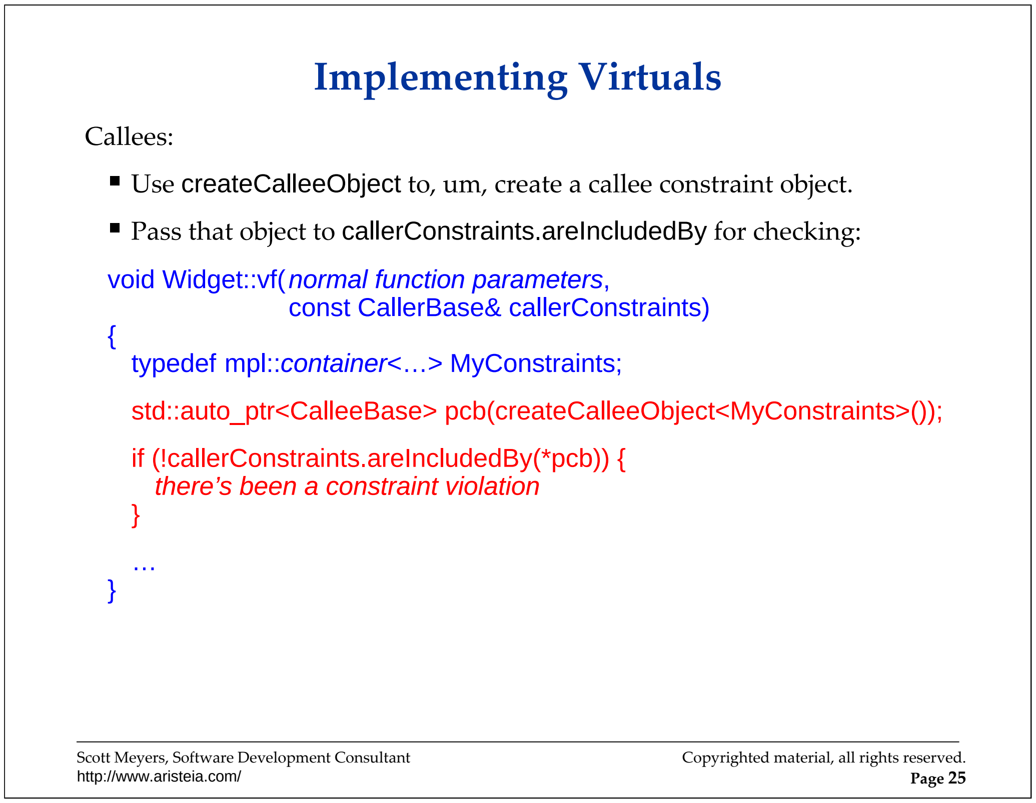


Calling Virtuals

Callers simply pass a temporary CallerConstraintsObject to the virtual:

```
template<typename CallerConstraints>
void f(Widget& w)
{
  // the usual stuff
  typedef mpl::container<…> MyConstraints;
  BOOST_MPL_ASSERT(( includes<MyConstraints, CallerConstraints> ));

  …
  w.vf( normal parameters,
        CallerConstraintsObject<MyConstraints>());
  …
}
```

See the source code for CallerConstraintsObject's implementation.



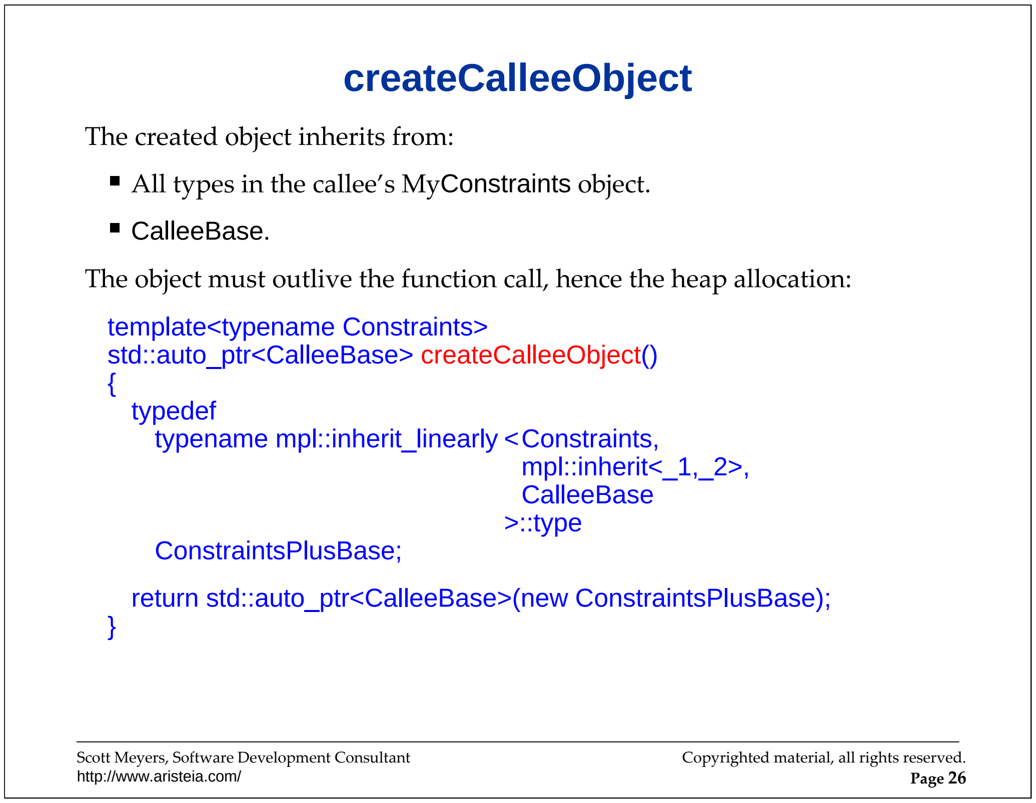
Implementing Virtuals

Callees:

- Use createCalleeObject to, um, create a callee constraint object.
- Pass that object to callerConstraints.areIncludedBy for checking:

```
void Widget::vf(normal function parameters,
                const CallerBase& callerConstraints)
{
  typedef mpl::container<…> MyConstraints;

  std::auto_ptr<CalleeBase> pcb(createCalleeObject<MyConstraints>());

  if (!callerConstraints.areIncludedBy(*pcb)) {
    there's been a constraint violation
  }

  …
}
```

See the notes on the next page for how the use of auto_ptr can be eliminated



createCalleeObject

The created object inherits from:

- All types in the callee’s MyConstraints object.
- CalleeBase.

The object must outlive the function call, hence the heap allocation:

```
template<typename Constraints>
std::auto_ptr<CalleeBase> createCalleeObject()
{
  typedef
    typename mpl::inherit_linearly <Constraints,
                                     mpl::inherit<_1,_2>,
                                     CalleeBase
                                    >::type
    ConstraintsPlusBase;

  return std::auto_ptr<CalleeBase>(new ConstraintsPlusBase);
}
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 26

Per Eric Niebler’s suggestion during the talk, the auto_ptr return type can be eliminated by returning an object by value and requiring that the caller bind it to a reference to const. That would result in a function that could be defined like this:

```
template<typename Constraints>
typename mpl::inherit_linearly<Constraints, mpl::inherit<_1,_2>, CalleeBase>::type
createCalleeObject()
{
    typedef  typename mpl::inherit_linearly<Constraints,
                                             mpl::inherit<_1,_2>,
                                             CalleeBase
                                             >::type
             ConstraintsPlusBase;

    return ConstraintsPlusBase();
}
```

Callers would then be expected to use it like this:

```
const CalleeBase& calleeConstraintObject = createCalleeObject<MyConstraints>();

if (!callerConstraints.areIncludedBy(calleeConstraintObject)) {
    there’s been a constraint violation
}
```

Virtual Function Summary

Two choices:

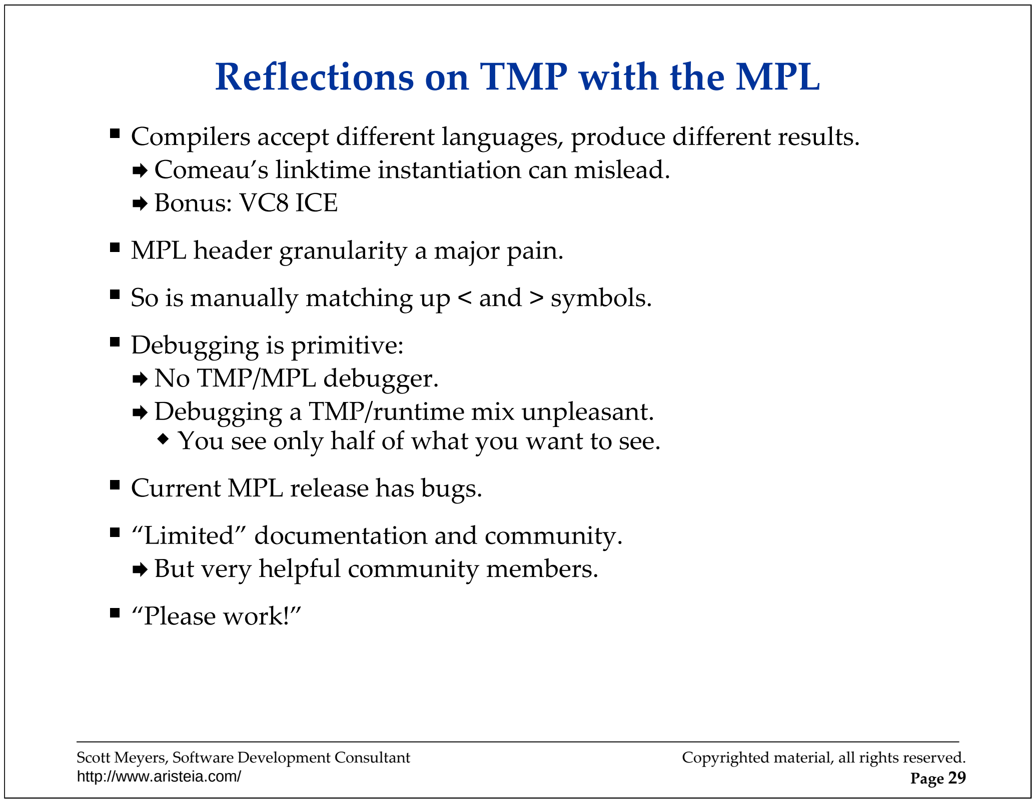
- NVI.
- RTTI:
  - Virtuals declare an extra CallerBase parameter.
  - Callers use CallerConstraintsObject<MyConstraints>() as that parameter.
  - Callees:
    - Call createCalleeObject<MyConstraints>() and
    - Pass the result to the caller's areIncludedBy function.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/
Copyrighted material, all rights reserved.
Page 27

Remaining Problem

How deal with operators?

- Can’t pass additional runtime parameters.
- Explicitly passing template parameters destroys the syntax.

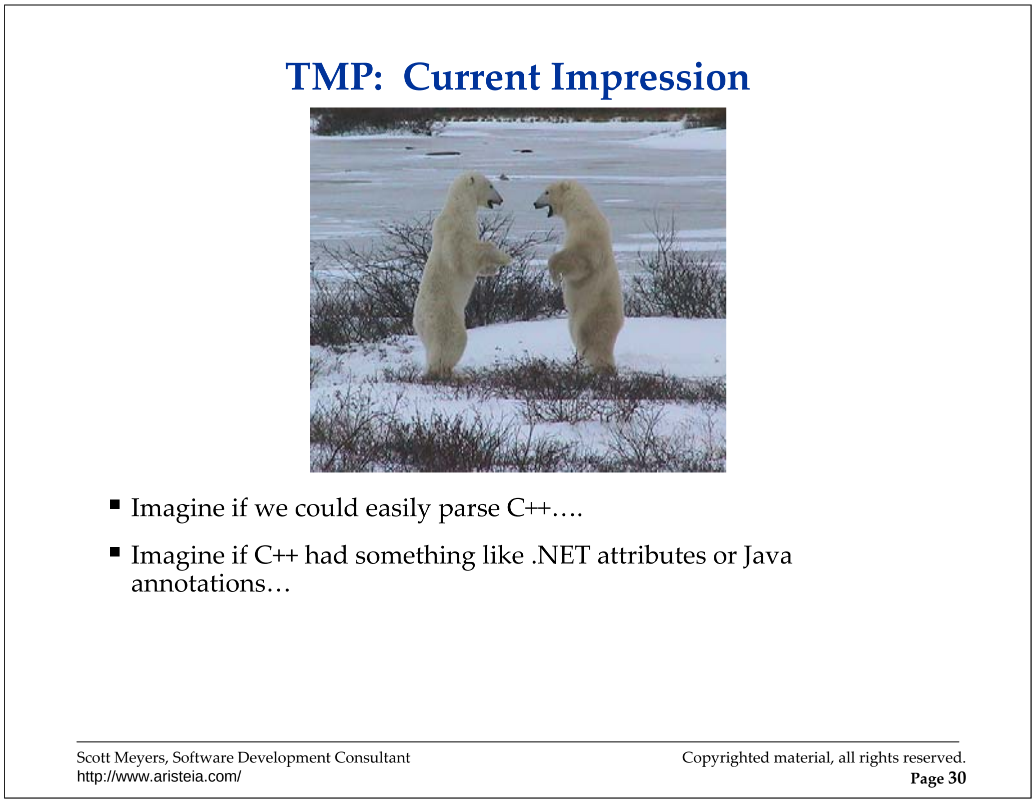


Reflections on TMP with the MPL

- Compilers accept different languages, produce different results.
  - Comeau's linktime instantiation can mislead.
  - Bonus: VC8 ICE
- MPL header granularity a major pain.
- So is manually matching up < and > symbols.
- Debugging is primitive:
  - No TMP/MPL debugger.
  - Debugging a TMP/runtime mix unpleasant.
    - You see only half of what you want to see.
- Current MPL release has bugs.
- "Limited" documentation and community.
  - But very helpful community members.
- "Please work!"

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 29



TMP: Current Impression

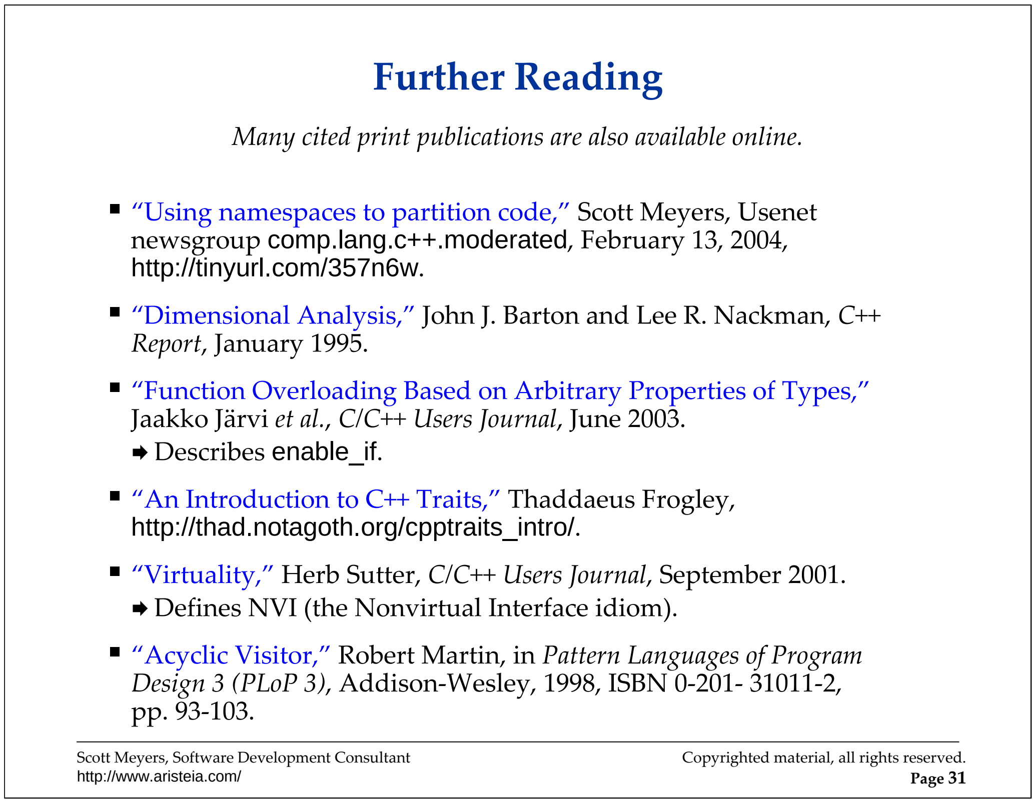
- Imagine if we could easily parse C++….
- Imagine if C++ had something like .NET attributes or Java annotations…

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 30

The image is motivated by Alan Cooper's notion of "Dancing Bearware" in *The Inmates are Running the Asylum* (Sams, 1999).

At the time I put this slide together, I was under the impression that attributes in .NET could be used to execute user code during compilation, and that could be used to achieve the same goal I described in this talk. One person came up to me after the talk and told me that .NET attributes were not that powerful. If that is the case, perhaps dancing bearware is all we have for attacking this problem.



Further Reading

Many cited print publications are also available online.

- "Using namespaces to partition code," Scott Meyers, Usenet newsgroup comp.lang.c++.moderated, February 13, 2004, http://tinyurl.com/357n6w.
- "Dimensional Analysis," John J. Barton and Lee R. Nackman, *C++ Report*, January 1995.
- "Function Overloading Based on Arbitrary Properties of Types," Jaakko Järvi *et al.*, *C/C++ Users Journal*, June 2003.
  - ➡ Describes enable_if.
- "An Introduction to C++ Traits," Thaddaeus Frogley, http://thad.notagoth.org/cpptraits_intro/.
- "Virtuality," Herb Sutter, *C/C++ Users Journal*, September 2001.
  - ➡ Defines NVI (the Nonvirtual Interface idiom).
- "Acyclic Visitor," Robert Martin, in *Pattern Languages of Program Design 3 (PLoP 3)*, Addison-Wesley, 1998, ISBN 0-201- 31011-2, pp. 93-103.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

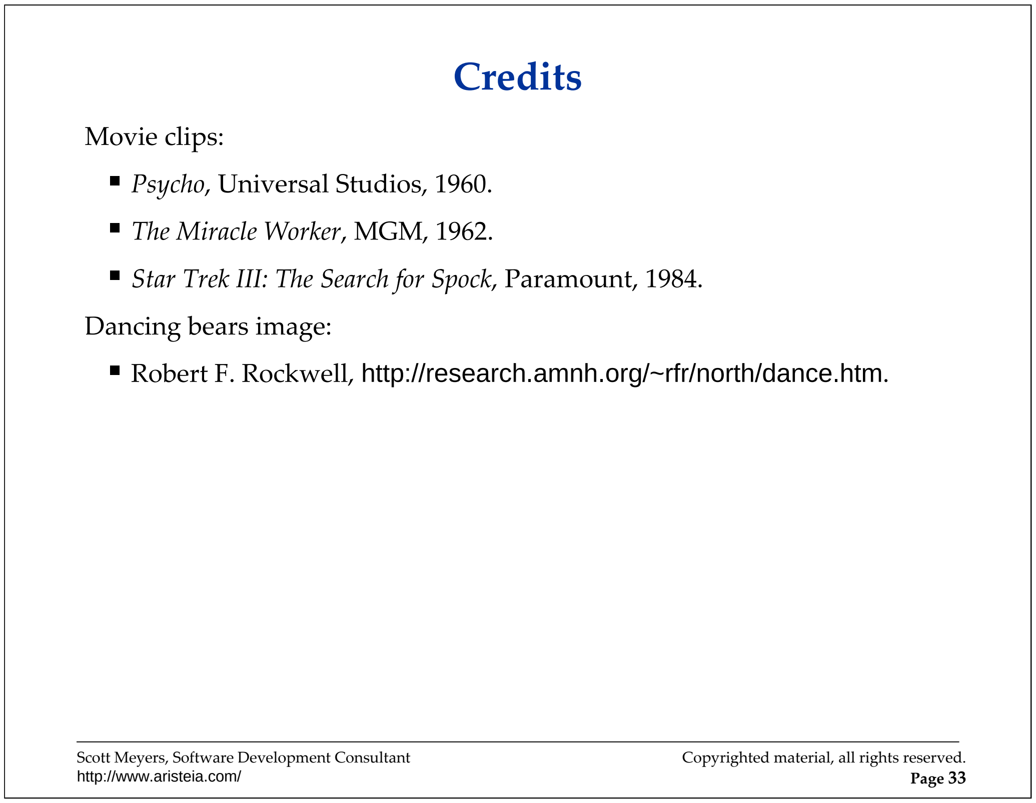
Copyrighted material, all rights reserved.
Page 31

Having createCalleeObject return a temporary is practical, because of the C++ rule that extends the life of a temporary when it is bound to a reference-to-const. My inspiration for use of this idea was:

> "Change the Way You Write Exception-Safe Code – Forever," Andrei Alexandrescu and Petru Marginean, *C/C++ Users Journal*, December 2000.

Further Reading

- *C++ Template Metaprogramming*, David Abrahams and Aleksey Gurtovoy, Addison-Wesley, 2004, ISBN 0-321-22725-5.
- *The Boost MPL Library*, http://www.boost.org/libs/mpl/doc/index.html.



Credits

Movie clips:

- *Psycho*, Universal Studios, 1960.
- *The Miracle Worker*, MGM, 1962.
- *Star Trek III: The Search for Spock*, Paramount, 1984.

Dancing bears image:

- Robert F. Rockwell, http://research.amnh.org/~rfr/north/dance.htm.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 33