# Enforcing Code Feature Requirements in C++

**Scott Meyers, Ph.D.**
Software Development Consultant

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

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





## const as a Code Feature

const member functions are treated like they offer a special *code feature.*

- Feature semantics lead to constraints:
  - Unconstrained code → constrained code: always okay.
    - E.g., non-const code → const code.
  - Constrained code → unconstrained code: okay only with permission.
    - E.g., a cast.

const is actually a data feature, but we'll ignore that.

Compilers enforce const-related constraints.

## Other Code Features

Analogous features are easy to imagine:

- Thread-safe.
- Exception-safe.
- Portable.
- Side-effect free.

**Goal: automatic enforcement of constraints for arbitrary code features.**





## Types as Features

Basic idea:

- Features are represented by UDTs.
  - ➧ Just like STL iterator categories.
- A function's features is represented by a set of types.
- Caller → callee is okay iff (caller features) ⊆ (callee features).
  - ➧ Use TMP to enforce this during compilation.

## The Boost 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 should be easier than "bare metal" TMP.





## Declaring Features

Features are represented by UDTs:

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

The MPL lets us create containers of these types:

```
typedef boost::mpl::vector<ThreadSafe, ExceptionSafe> TESafe;
```

## Declaring Features

Macros ease the busy work:

```
CREATE_CODE_FEATURES_4(ThreadSafe, ExceptionSafe, Portable, Reviewed)
```

- Creates types for ThreadSafe, ExceptionsSafe, Portable, Reviewed.
- Creates an MPL vector, AllCodeFeatures, containing them all.
  - ➡ We'll use this later.





## Declaring Functions' Features

Functions with code features declare a parameter for them:

- Parameter type is *MakeFeatures<FeatureContainer>::type*:

```
void f( int x, double y,
        MakeFeatures<TESafe>::type features)
{
  …                                    // normal function body
}
```

- Parameter has meaning only during compilation:
  - ➡ Declares which features the function offers.
  - ➡ Does nothing at runtime.

## Caller → Callee Feature Communication

Callers specify the features they need via a suitable call features object:

- Often they use their call features parameter.

```
typedef boost::mpl::vector<ThreadSafe, ExceptionSafe, Portable> TEPSafe;
void g(MakeFeatures<TEPSafe>::type features);              // declare g

void f(int x, double y, MakeFeatures<TESafe>::type features)
{
  g(features);                        // fine, g offers all features f needs
}
void g(MakeFeatures<TEPSafe>::type features)
{
  int xVal, yVal;
  …
  f(xVal, yVal, features);            // error!  f is missing Portable feature
  …
}
```





## Diagnostic from g++ 4

```
articlecode.cpp: In function 'void g(
  CodeFeatures::Features<
    boost::mpl::v_item<
      CodeFeatures::Portable
    , boost::mpl::v_item<
        CodeFeatures::ExceptionSafe
      , boost::mpl::v_item<
          CodeFeatures::ThreadSafe,boost::mpl::vector0<mpl_::na>
          , 0
          >, 0
        >, 0
      >
    >
  )':
articlecode.cpp:32: error: conversion from 'CodeFeatures::Features<
  boost::mpl::v_item<
    CodeFeatures::Portable
  , boost::mpl::v_item<
      CodeFeatures::ExceptionSafe
    , boost::mpl::v_item<
        CodeFeatures::ThreadSafe, boost::mpl::vector0<mpl_::na>, 0
        >, 0
      >, 0
    >
  >' to non-scalar type 'CodeFeatures::Features<
      boost::mpl::v_item<
        CodeFeatures::ExceptionSafe
      , boost::mpl::v_item<
          CodeFeatures::ThreadSafe, boost::mpl::vector0<mpl_::na>, 0
        >, 0
      >
    >' requested
```

## Diagnostic from VC9

```
articlecode.cpp(32) : error C2664: 'g' : cannot convert parameter 3 from
'CodeFeatures::Features<S>' to 'CodeFeatures::Features<S>'
  with
  [
    S=boost::mpl::vector3<CodeFeatures::ThreadSafe,CodeFeatures::Exception
Safe,CodeFeatures::Portable>
  ]
  and
  [
    S=boost::mpl::vector2<CodeFeatures::ThreadSafe,CodeFeatures::Exception
Safe>
  ]
  No user-defined-conversion operator available that can perform this
conversion, or the operator cannot be called
```





## Caller → Callee Feature Communication

- Callers can also create a new call features parameter:

```
void h()                                    // no CallFeatures parameter
{
  typedef mpl::container<…> NeededFeatures;
  int xVal, yVal;
  …
  f(xVal, yVal,                                      // okay if f offers all
    MakeFeatures<NeededFeatures>::type());           // features in
  …                                                  // NeededFeatures
}
```

## Relaxing Feature Requirements

Two analogues to const_cast for overriding feature requirements:

- Pass IgnoreFeatures as the CallFeatures object.
  - ➡ It "casts away" all required callee features.

```
void g(MakeFeatures<TEPSafe>::type features)              // as before
{
  int xVal, yVal;
  …
  f(xVal, yVal, IgnoreFeatures());          // fine, g's feature reqs ignored
  …
}
```

  - ➡ Implementation is just an empty container of features:

```
typedef MakeFeatures<mpl::vector<> >::type IgnoreFeatures;
```





## Relaxing Feature Requirements

- Use eraseVal to "remove" features from NeededFeatures.
  - ➡ TMP is functional, so the result is a new collection.

```
void g(MakeFeatures<TEPSafe>::type features)              // as before
{
  …
  typedef eraseVal<TEPSafe, Portable>::type RevisedFeatures;

  f(xVal, yVal,
    MakeFeatures<RevisedFeatures>::type());               // okay
  …
}
```

## eraseVal

No eraseVal in the MPL, but it's easy to implement:

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

For this application, eraseVal needed only as a workaround:

- mpl::set was broken in Boost 1.34 (current when I did this research).
- It's supposed to be fixed in 1.35.
  - ➧ Should be possible to use mpl::erase_key on mpl::set.





## Gratuitous Cute Niece Photograph

## Overloading on Feature Sets

Consider:

```
typedef mpl::vector<ThreadSafe, ExceptionSafe> TESafe;                  // as before
typedef mpl::vector<ThreadSafe, ExceptionSafe, Portable> TEPSafe;       // as before

void g(parameters, MakeFeatures<TESafe>::type);                         // call this g_TE
void g(parameters, MakeFeatures<TEPSafe>::type);                        // call this g_TEP

typedef boost::mpl::vector<ThreadSafe> TSafe;

void foo(parameters, MakeFeatures<TSafe>::type features)
{
  ...
  g(parameters, features);                                              // which g?
  ...
}
```

Both gs offer the needed features.

- **Policy: fewer unneeded features trumps more.**
  - ➡ Call $g_{TE}$.





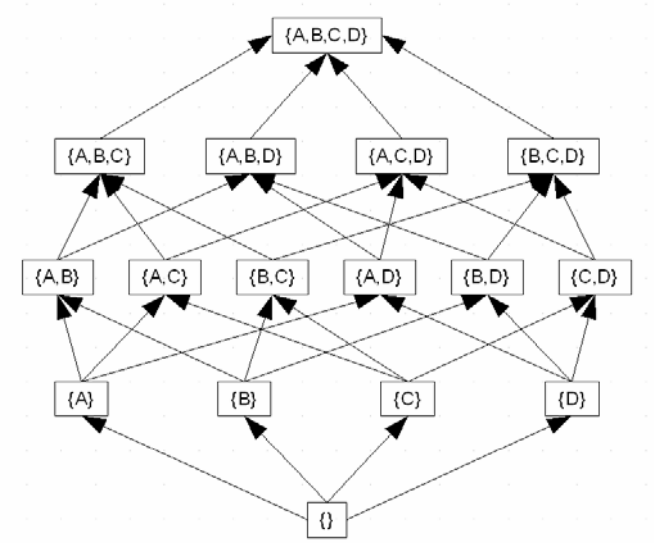
## Feature Set Conversion Rules

For feature set types $T_{needed}$ and $T_{offered}$:

- $T_{needed}$ converts to $T_{offered}$ only if $T_{offered}$ has all the features in $T_{needed}$.
- If more than one $T_{offered}$ is viable, fewer unneeded features trumps more.
  - ➡ Multiple equally good conversions  conversion is ambiguous.

## Conversion Rules via an Inheritance Hierarchy

Consider features A, B, C, and D:




*All inheritance is virtual.*

Challenge: How generate this hierarchy automatically?





## Observation #1

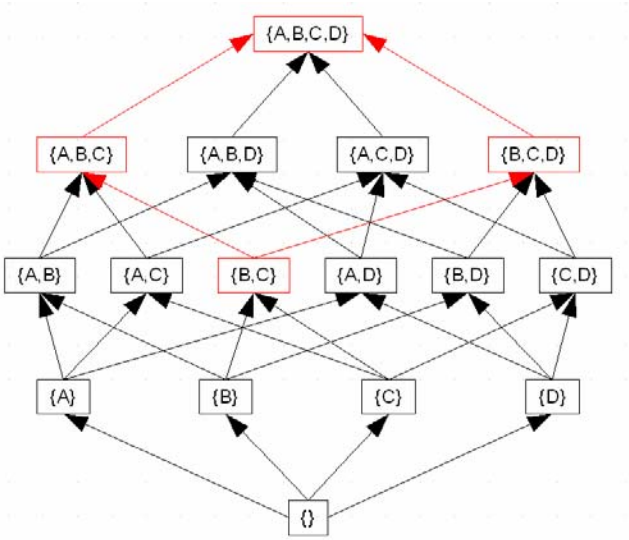
Feature sets are unordered:

- {A,B} ≡ {B,A}
- MakeFeatures<mpl::vector<A,B> >::type should behave the same as MakeFeatures<mpl::vector<B,A> >::type.

## Observation #2

Often, only part of the hierarchy is needed.

- Consider when $T_{needed}$ = {B,C}:



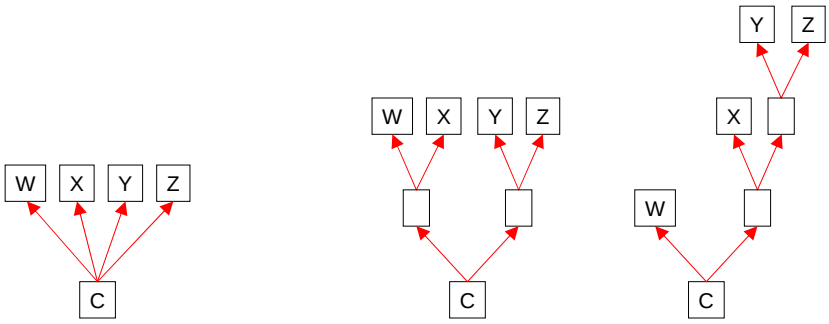





## Observation #3

Conversions with *n*-way inheritance can be achieved via dual inheritance.

- These all support implicit conversion from C to W, X, Y, and Z:




*n* base classes/level

≤ 2 base classes/level

## Implementing MakeFeatures

```
namespace mpl = boost::mpl;
using mpl::_1;
using mpl::_2;

template<typename S, typename T>                    // compute index of T in S
struct IndexOf:
  mpl::distance<typename mpl::begin<S>::type,
                typename mpl::find<S, T>::type>
{};

template<typename Unordered>                        // order contents of Unordered
struct Order:
  mpl::sort<Unordered,
            mpl::less< IndexOf<AllCodeFeatures, _1>,
                       IndexOf<AllCodeFeatures, _2> > >
{};

template<typename CF>                               // CF = "Container of Features"
struct MakeFeatures {
  typedef
    Features<typename mpl::copy<typename Order<CF>::type,
                                mpl::back_inserter<mpl::vector0<> > >::type>
  type;
};
```





## Some Useful Code

Virtually inherit from two base classes:

```
template<typename Base1, typename Base2>
struct VirtualInherit : virtual Base1, virtual Base2
{};
```

- Used to break *n*-way inheritance into hierarchies of 2-way inheritance.

Calculate the difference of two feature sets:

```
// Difference<S1,S2>::type is S1-S2
template<typename S1, typename S2>
struct Difference:
  mpl::remove_if<S1, mpl::contains<S2, _> >
{};
```

## Implementing Features

```
template<typename Present, typename Omitted>
struct GetFeaturesBases:
  mpl::transform<Omitted, MakeFeatures<mpl::push_back<Present, _1> > >
{};

template<typename S>
struct Features:                          // inherits from all supersets of S
  virtual mpl::fold<
    typename GetFeaturesBases<S,
                              typename Difference<AllCodeFeatures, S>::type
                             >::type,
    mpl::empty_base,
    VirtualInherit<_, _>
  >::type
{};
```





## Virtual Functions

Virtual overrides *should* be able to offer extra features, but C++ says no:

```
class Base {
public:
  typedef mpl::vector<ThreadSafe, Reviewed> BaseFeatures;

  virtual void vf(int x, MakeFeatures<BaseFeatures>::type features);
  …
};

class Derived: public Base {
public:
  typedef mpl::vector<ThreadSafe, Reviewed, Portable> DerFeatures;

  virtual void vf( int x,                                          // doesn't
                   MakeFeatures<DerFeatures>::type features);  // override
  ...                                                              // Base::vf!
};
```

Oh for contravariant parameter types!

## Virtual Functions

We can fake it via overloading in the derived class:

```
class Derived: public Base {
public:
  typedef mpl::vector<ThreadSafe, Reviewed, Portable> DerFeatures;

  virtual void vf(int x,                                          // override
                  MakeFeatures<BaseFeatures>::type features)      // base::vf
  {
    return vf(x, MakeFeatures<DerFeatures>::type());              // call other
  }                                                               //  vf

  virtual void vf(int x,                                          // as before
                  MakeFeatures<DerFeatures>::type features);
  ...
};
```

Situation is akin to that for covariant return types:

- Users of base class interface get the "basic" interface.
- Users of derived class interface get the "enhanced" interface.

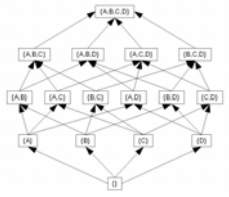




## Performance

- **Compilation:**
  - ➡ *Much* slower.
    - ◆ Test programs (<100 lines, excluding headers) took tens of seconds.
  - ➡ VC9 yielded ICEs on features sets with more than 5 features.
- **Runtime:**
  - ➡ sizeof(MakeFeatures<*Features*>::type) can be large:

| Features in Feature Set Object | gcc 4.1.1 | Visual C++ 9 | Comeau 4.3.9 |
|---|---|---|---|
| 0 | 64 | 388 | 7672 |
| 1 | 32 | 164 | 1884 |
| 2 | 16 | 68 | 452 |
| 3 | 8 | 28 | 108 |
| 4 | 4 | 12 | 28 |
| 5 | 4 | 4 | 8 |



    - ◆ In theory, compilers could optimize such objects away.

Current implementation is proof-of-concept only.

- Feature set objects are passed by value, not by pointer or reference!

## Open Issues

- **How deal with operators?**
  - ➧ Can't pass additional parameters...
- **How eliminate need for AllCodeFeatures?**
  - ➧ Possible to implement compile-time self-registration of features?
- **How improve feature constraint violation messages?**
  - ➧ Maybe something like STLFilt?
- **How specify feature constraints for groups of functions?**
  - ➧ E.g., per-class or per-namespace.
- **How improve performance?**
  - ➧ Dual header sets, a slow "real" one and a fast "no-op" one?





## Acknowledgments

Important code or ideas contributed by:

- **Members of Boost mailing list**
  - ➧ Especially Steven Watanabe
- Herb Sutter
- Andrei Alexandrescu

## Further Reading

*Many cited print publications are also available online.*


- "Enforcement of Code Feature Requirements in C++," Scott Meyers, submitted for publication, May 2008.
- "Re: [mpl] Hierarchy Generation," Steven Watanabe, Boost User's Mailing List, February 25, 2008.
- "Thoughts on Scott's 'Red Code / Green Code' Talk," Herb Sutter, May 6, 2007, http://herbsutter.spaces.live.com/blog/cns!2D4327CC297151BB!207.entry.
- *The Boost MPL Library*, http://www.boost.org/libs/mpl/doc/index.html.
- *C++ Template Metaprogramming*, David Abrahams and Aleksey Gurtovoy, Addison-Wesley, 2004, ISBN 0-321-22725-5.






## Please Note

Scott Meyers offers consulting services in all aspects of the design and implementation of C++ software systems. For details, visit his web site:

http://www.aristeia.com/

Scott also offers a mailing list to keep you up to date on his professional publications and activities. Read about the mailing list at:

http://www.aristeia.com/MailingList/