Boost.Generic: Concepts without Concepts

Matt Calabrese

Overview

- What is Generic Programming?
- The End of C++0x Concepts
- Substitution Failure Is Not an Error
- A Brief History of Boost.Generic
- The Necessary Tools
- Built-in Concepts and Asserts
- Creating Concepts
- Creating Concept Maps
- Future Direction
- Questions

What is Generic Programming?

You Tell Me

What is Generic Programming?

- Programming Paradigm
- Stepanov and Musser
- STL, BGL, Boost.GIL, etc.
- Algorithm-Centric
- Lifting
- Requirements/Constraints
- Multi-Sorted Algebras
- Refinement
- Concept Mapping
- Archetypes
- Concept-Based Overloading

The End of C++0x Concepts

Can't We All Just Get Along?

The End of C++0x Concepts

Indiana Proposal

- IU – Siek, Gregor, Garcia, Willcock, Järvi, Lumsdaine
- Explicit Concepts by Default
- Allow “Auto” Concepts
- Concept Maps

Texas Proposal

- Texas A&M – Bjarne Stroustrup, Gabriel Dos Rei
- All Concepts are Automatic
- No Explicit Concept Maps

Substitution Failure Is Not an Error

Or Is It!?*

*No, really, I'm not lying. It's not an error.

Substitution Failure Is Not an Error

What is SFINAE?

```
int negate(int i) { return -i; }

template <class F>
typename F::result_type negate(const F& f) { return -f(); }

int neg1 = negate( 1 );
```


Substitution Failure Is Not an Error

Boost.Enable_If

```
template< bool B, class T = void >
struct enable_if_c {
  typedef T type;
};

template< class T >
struct enable_if_c<false, T> {}

template< bool B, class T = void >
struct disable_if_c : enable_if_c< !b, T > {};

template< class T >
typename enable_if< has_trivial_destructor< T >::value >::type
foo( T& bar ) { … }

template< class T >
typename disable_if< has_trivial_destructor< T >::value >::type
foo( T& bar ) { … }
```


A Brief History of Boost.Generic

Or: How I Learned to Stop Worrying
and
Love the Preprocessor

A Brief History of Boost.Generic

It all began with C++0x “auto” functions...

```
template< class L, class R >
??? operator -( L lhs, R rhs )
{
  return lhs + -rhs;
}
```

What should the return type be?

A Brief History of Boost.Generic

It all began with C++0x "auto" functions...

```
template< class L, class R >
auto operator -( L lhs, R rhs ) -> decltype( lhs + -rhs )
{
  return lhs + -rhs;
}
```


A Brief History of Boost.Generic

It all began with C++0x “auto” functions...

```
template< class L, class R >
auto operator -( L lhs, R rhs ) -> decltype( lhs + -rhs )
{
  return lhs + -rhs;
}
```

But isn't this redundant?

A Brief History of Boost.Generic

It all began with C++0x “auto” functions...

```
template< class L, class R >
auto operator -( L lhs, R rhs ) -> decltype( lhs + -rhs )
{
  return lhs + -rhs;
}
```

But isn't this redundant?

Lambdas can deduce return types without redundancy.

```
auto minus = []( L lhs, R rhs )
{
  return lhs + -rhs;
};
```


A Brief History of Boost.Generic

Ahh... much better!

```
template< class L, class R >
BOOST_AUTO_FUNCTION( operator -( L lhs, R rhs ) )
(
  return lhs + -rhs
)
```

Macros save the day!

A Brief History of Boost.Generic

Ahh... much better!

```
template< class L, class R >
BOOST_AUTO_FUNCTION( operator -( L lhs, R rhs ) )
(
  return lhs + -rhs
)
```

Macros save the day!

But how would we use enable_if here?*

*In the world of C++0x before discovering the new way to use enable_if

A Brief History of Boost.Generic

More complicated macros, of course!

```
template< class L, class R >
BOOST_AUTO_FUNCTION( operator -( L const& lhs, R const& rhs )
                   , if ( is_vector_udt< L > )
                        ( is_vector_udt< R > )
                   )
(
  return lhs + -rhs
)
```


A Brief History of Boost.Generic

```
template< class L, class R >
BOOST_AUTO_FUNCTION( operator -( L const& lhs, R const& rhs )
                   , if ( is_vector_udt< L > )
                        ( is_vector_udt< R > )
                   , try ( lhs + rhs )
                         ( -rhs )
                   , if typename ( L::value_type )
                   )
(
  return lhs + -rhs
)
```

Okay, stop it already!

A Brief History of Boost.Generic

What if we used these ideas to make a macro for specifying concepts...

```
[utility.arg.requirements]
...In these tables, T is an object or reference type to be supplied by a C++ program
instantiating a template; a, b, and c are values of type (possibly const) T...
```

Table 17 — `EqualityComparable` requirements [**equalitycomparable**]

Expression	Return type	Requirement
`a == b`	convertible to `bool`	`==` is an equivalence relation, that is, it has the following properties: — For all `a`, `a == a`. — If `a == b`, then `b == a`. — If `a == b` and `b == c`, then `a == c`.

A Brief History of Boost.Generic

What if we used these ideas to make a macro for specifying concepts...

[utility.arg.requirements]
...In these tables, T is an object or reference type to be supplied by a C++ program instantiating a template; a, b, and c are values of type (possibly const) T...

Table 17 — EqualityComparable requirements [equalitycomparable]

Expression	Return type	Requirement
`a == b`	convertible to `bool`	`==` is an equivalence relation, that is, it has the following properties: — For all `a`, `a == a`. — If `a == b`, then `b == a`. — If `a == b` and `b == c`, then `a == c`.

```
BOOST_GENERIC_AD_HOC_CONCEPT
( (EqualityComparable)( (class) T )
, ( for ( (T const) a )
        ( (T const) b )
  )
, ( try ( a == b ) )
, ( if ( is_convertible< decltype( a == b ), bool > ) )
)
```

A Brief History of Boost.Generic

What if we used these ideas to make a macro for specifying concepts...

```
[utility.arg.requirements]
...In these tables, T is an object or reference type to be supplied by a C++ program
instantiating a template; a, b, and c are values of type (possibly const) T...
```

Table 17 — EqualityComparable requirements [equalitycomparable]

Expression	Return type	Requirement
`a == b`	convertible to `bool`	`==` is an equivalence relation, that is, it has the following properties: — For all `a`, `a == a`. — If `a == b`, then `b == a`. — If `a == b` and `b == c`, then `a == c`.

```
BOOST_GENERIC_AD_HOC_CONCEPT
( (EqualityComparable)( (class) T )
, ( for ( (T const) a )
        ( (T const) b )
  )
, ( try ( a == b ) )
, ( if ( is_convertible< decltype( a == b ), bool > ) )
)
```

The Necessary Tools

What Can Be Accomplished
and How

The Necessary Tools

Overall Goals

- Represent Concepts as Closely as Possible to N2914
- Support the Major Features of Concepts
- Keep the Error Messages Simple (Simpler than BCCL)
- Automatically Generate Archetypes
- Support Concept Overloading
- Allow a Choice of Backends
- Implement the Standard Concepts
- Minimize the Pain Inflicted on Compilers

The Necessary Tools

Overall Goals

- Represent Concepts as Closely as Possible to N2914
- Support the Major Features of Concepts
- Keep the Error Messages Simple (Simpler than BCCL)
- Automatically Generate Archetypes
- Support Concept Overloading
- Allow a Choice of Backends
- Implement the Standard Concepts
- Minimize the Pain Inflicted on Compilers (Just Kidding)

The Necessary Tools

C++ 0x Syntax

```
concept ArithmeticLike<typename T>
  : Regular<T>, LessThanComparable<T>, HasUnaryPlus<T>, HasNegate<T>,
    HasPlus<T, T>, HasMinus<T, T>, …
{
  explicit T::T(intmax_t);
  explicit T::T(uintmax_t);
  explicit T::T(long double);
  requires Convertible<HasUnaryPlus<T>::result_type, T>;
  …
}
```


The Necessary Tools

C++ 0x Syntax

```
concept ArithmeticLike<typename T>
  : Regular<T>, LessThanComparable<T>, HasUnaryPlus<T>, HasNegate<T>,
    HasPlus<T, T>, HasMinus<T, T>, …
{
  explicit T::T(intmax_t);
  explicit T::T(uintmax_t);
  explicit T::T(long double);
  requires Convertible<HasUnaryPlus<T>::result_type, T>;
  …
}
```

Hypothetical Macro Syntax

```
BOOST_GENERIC_CONCEPT
( (ArithmeticLike)( (typename) T )
, ( public Regular<T>, LessThanComparable<T>, HasUnaryPlus<T>, HasNegate<T>,
           HasPlus<T, T>, HasMinus<T, T>, …
  )
, ( explicit (this(T))( (intmax_t) ) )
, ( explicit (this(T))( (uintmax_t) ) )
, ( explicit (this(T))( (long double) ) )
, ( requires Convertible<typename HasUnaryPlus<T>::result_type, T> )
, …
)
```


The Necessary Tools

Just what can be accomplished (not a complete list)?

Language-Level Concepts	Library-Based Concept Support
Associate Types	Yes
Associate Functions	Yes
Multi-type Concepts	Yes
Concept Maps	Yes
Archetypes	Yes
Concept Refinement	???
Typename Deduction	???
Concept-Based Overloading	???

The Necessary Tools

List of Types Across a Translation Unit

```
template< unsigned Val = 256 > struct num_elem : num_elem< Val - 1 > {};
template<> struct num_elem< 0 > {};

template< class Tag, class ThisElem = void, class OtherHolder = void >
struct type_seq_holder
{
  static unsigned const value = OtherHolder::value + 1;
  static num_elem< value + 1 > next_index();
  typedef typename push_back< typename OtherHolder::type_seq, ThisElem >::type
          type_seq;
};

template< class Tag >
struct type_seq_holder< Tag >
{
  static unsigned const value = 0;
  static num_elem< 1 > next_index();
  typedef vector<> type_seq;
};

template< class Tag > type_seq_holder< Tag > get_type_seq_holder( Tag, num_elem< 0 > const& );

#define ADD_TO_LIST( list_tag, new_elem )\
type_seq_holder< list_tag, new_elem, decltype( get_type_seq_holder( list_tag(), num_elem<>() ) ) >\
get_type_seq_holder\
( list_tag, decltype( get_type_seq_holder(list_tag(), num_elem<>() ).next_index() ) const& );

#define GET_LIST( list_tag )\
identity< decltype( get_type_seq_holder( list_tag(), num_elem<>() ) ) >::type::type_seq
```

The Necessary Tools

Just what can be accomplished (not a complete list)?

Language-Level Concepts	Library-Based Concept Support
Associate Types	Yes
Associate Functions	Yes
Multi-type Concepts	Yes
Concept Maps	Yes
Archetypes	Yes
Concept Refinement	Yes
Typename Deduction	???
Concept-Based Overloading	???

The Necessary Tools

Typename Deduction

```
auto concept HasFind<typename T> {
  typename key_type = typename T::key_type;
  typename mapped_type;
  std::pair< key_type, mapped_type > find( T const&, key_type const& );
}

struct container { typedef int key_type; };
std::pair< int, float > find( container, int );
```


The Necessary Tools

Typename Deduction

```
auto concept HasFind<typename T> {
  typename key_type = typename T::key_type;
  typename mapped_type;
  std::pair< key_type, mapped_type > find( T const&, key_type const& );
}

struct container { typedef int key_type; };
std::pair< int, float > find( container, int );
```


The Necessary Tools

Typename Deduction

```
auto concept HasFind<typename T> {
  typename key_type = typename T::key_type;
  typename mapped_type;
  std::pair< key_type, mapped_type > find( T const&, key_type const& );
}

struct container { typedef int key_type; };
std::pair< int, float > find( container, int );
```

Very Common With Auto-Concepts

```
auto concept HasDereference<typename T> {
  typename result_type;
  result_type operator*(T&);
  result_type operator*(T&&);
}
```


The Necessary Tools

Typename Deduction

14.10.2.2 Associated type and template definitions **[concept.map.assoc]**

"…A concept map member that satisfies an associated type or class template requirement can be implicitly defined using template argument deduction (14.9.2) with one or more associated function requirements (14.10.2.1), if the associated type or class template requirement does not have a default value. The definition of the associated type or class template is determined using the rules of template argument deduction from a type (14.9.2.5).

– Let P be the return type of an associated function after substitution of the concept's template parameters specified by the concept map with their template arguments, and where each undefined associated type and associated class template has been replaced with a newly invented type or template template parameter, respectively.

– Let A be the return type of the seed in the associated function candidate set corresponding to the associated function.

If the deduction fails, no concept map members are implicitly defined by that associated function. If the results of deduction produced by different associated functions yield more than one possible value, that associated type or class template is not implicitly defined…"

The Necessary Tools

Typename Deduction

```
BOOST_GENERIC_AUTO_CONCEPT
( (HasFind)( (typename) T )
, ( typename key_type, typename T::key_type )
, ( typename mapped_type )
, ( (std::pair< key_type, mapped_type >)(find)( (T const&), (key_type const&) ) )
)
```


The Necessary Tools

Typename Deduction

```
BOOST_GENERIC_AUTO_CONCEPT
( (HasFind)( (typename) T )
, ( typename key_type, typename T::key_type )
, ( typename mapped_type )
, ( (std::pair< key_type, mapped_type >)(find)( (T const&), (key_type const&) ) )
)
```

The Necessary Tools

Typename Deduction

```
BOOST_GENERIC_AUTO_CONCEPT
( (HasFind)( (typename) T )
, ( typename key_type, typename T::key_type )
, ( typename mapped_type )
, ( (std::pair< key_type, mapped_type >)(find)( (T const&), (key_type const&) ) )
)
```

Exploit Template Argument Deduction

```
typedef typename T::key_type key_type;

struct dummy_type {};

identity< dummy_type > deduce( ... );

template< class mapped_type >
identity< mapped_type > deduce( identity< std::pair< key_type, mapped_type > > );

typedef decltype( deduce( identity< decltype( find( declval<T const&>(), declval<key_type const&>() ) ) >() ) )
        deduce_result;

typedef typename deduce_result::type mapped_type;
```

The Necessary Tools

Just what can be accomplished (not a complete list)?

Language-Level Concepts	Library-Based Concept Support
Associate Types	**Yes**
Associate Functions	**Yes**
Multi-type Concepts	**Yes**
Concept Maps	**Yes**
Archetypes	**Yes**
Concept Refinement	**Yes**
Typename Deduction	**Yes**
Concept-Based Overloading	???

The Necessary Tools

Can we use a technique similar to tag dispatching?

```
template< class It, class DiffT >
void advance( It& it, DiffT offset )
{
  typedef typename std::iterator_traits< It >::iterator_category category;

  advance_impl( it, offset, category() );
}

template< class It, class DiffT >
void advance_impl( It& it, DiffT offset, std::input_iterator_tag ) /**/

template< class It, class DiffT >
void advance_impl( It& it, DiffT offset, std::bidirectional_iterator_tag ) /**/

template< class It, class DiffT >
void advance_impl( It& it, DiffT offset, std::random_access_iterator_tag ) /**/
```

We can start by automatically creating tags that are related by inheritance...

The Necessary Tools

Just what can be accomplished (not a complete list)?

Language-Level Concepts	Library-Based Concept Support
Associate Types	Yes
Associate Functions	Yes
Multi-type Concepts	Yes
Concept Maps	Yes
Archetypes	Yes
Concept Refinement	Yes
Typename Deduction	Yes
Concept-Based Overloading	Almost

Built-in Concepts and Asserts

For the Lazy Programmers Who Want Me to Do All of the Work

Built-in Concepts and Asserts

Which Concepts Are Currently Implemented?

- <concepts> 78/78
- <container_concepts> 0/20
- <iterator_concepts> 8/8
- <memory_concepts> 0/6
- [concept.support] 23/23

Built-in Concepts and Asserts

The Basic Header Structure

```
// Include all built-in concepts
#include <boost/generic/std_concept.hpp>

// Include all iterator concepts (akin to <iterator_concepts>)
#include <boost/generic/std_concept/iterator_concepts.hpp>

// Include just the forward iterator concept and its dependencies
#include <boost/generic/std_concept/iterator_concepts/forward_iterator.hpp>

// Etc.

// Include all assert macros
#include <boost/generic/assert.hpp>
```

Concepts are based on N2914

Built-in Concepts and Asserts

Concept Asserts

```
BOOST_GENERIC_ASSERT( HasPlus< int*, int> );

BOOST_GENERIC_ASSERT_NOT( HasPlus< int*, int* > );

// Triggers a static_assert
BOOST_GENERIC_ASSERT( HasPlus< int*, int* > )
```

Built-in Concepts and Asserts

Concept Asserts

```
BOOST_GENERIC_ASSERT( HasPlus< int*, int> );

BOOST_GENERIC_ASSERT_NOT( HasPlus< int*, int* > );

// Triggers a static_assert
BOOST_GENERIC_ASSERT( HasPlus< int*, int* > )
```

```
has_plus.cpp:14:177: error: static assertion failed: "requires HasPlus< int*, int* >"
has_plus.cpp:10:0:
has_plus.hpp: In instantiation of 'boost::generic::std_concept::HasPlus<int*, int*>':
has_plus.cpp:14:543:   instantiated from here
has_plus.hpp:20:10224: error: static assertion failed: "typename \'result_type\' was not
explicitly satisfied and cannot be deduced."
has_plus.hpp: In instantiation of 'boost::generic::std_concept::HasPlus<int*, int*>':
has_plus.cpp:14:543:   instantiated from here
has_plus.hpp:20:10691: error: static assertion failed: "requires result_type operator +( const
T& , const U& )"
```


Creating Concepts

Enough already, I want to make my own!

Creating Concepts

Starting with the Basics...

```
namespace boost { namespace generic {

concept Foo<typename T> {}

} }
```


Creating Concepts

Starting with the Basics...

```
namespace boost { namespace generic {

concept Foo<typename T> {}

} }
```

```
BOOST_GENERIC_CONCEPT
( ( namespace boost, generic )
, (Foo)( (typename) T )
)
```


Creating Concepts

Starting with the Basics...

```
namespace boost { namespace generic {

concept Foo<typename T> {}

} }
```

```
BOOST_GENERIC_CONCEPT
( ( namespace boost, generic )
, (Foo)( (typename) T )
)
```


Creating Concepts

Starting with the Basics...

```
namespace boost { namespace generic {

concept Foo<typename T> {}

} }
```

```
BOOST_GENERIC_CONCEPT
( ( namespace boost, generic )
, (Foo)( (typename) T )
)
```


Creating Concepts

Starting with the Basics...

```
namespace boost { namespace generic {

concept Foo<typename T> {}

} }
```

```
BOOST_GENERIC_CONCEPT
( ( namespace boost, generic )
, (Foo)( (typename) T )
)
```


Creating Concepts

Starting with the Basics...

```
auto concept IdentityOf<typename T> {
  typename type = T;
  requires SameType<type, T>;
}
```


Creating Concepts

Starting with the Basics...

```
auto concept IdentityOf<typename T> {
  typename type = T;
  requires SameType<type, T>;
}
```

```
BOOST_GENERIC_AUTO_CONCEPT
( (IdentityOf)( (typename) T )
, ( typename type, T )
, ( requires SameType<type, T> )
)
```


Creating Concepts

Starting with the Basics...

```
auto concept IdentityOf<typename T> {
  typename type = T;
  requires SameType<type, T>;
}
```

```
BOOST_GENERIC_AUTO_CONCEPT
( (IdentityOf)( (typename) T )
, ( typename type, T )
, ( requires SameType<type, T> )
)
```

Creating Concepts

Starting with the Basics...

```
auto concept IdentityOf<typename T> {
  typename type = T;
  requires SameType<type, T>;
}
```

```
BOOST_GENERIC_AUTO_CONCEPT
( (IdentityOf)( (typename) T )
, ( typename type, T )
, ( requires SameType<type, T> )
)
```

Creating Concepts

Starting with the Basics...

```
auto concept IdentityOf<typename T> {
  typename type = T;
  requires SameType<type, T>;
}
```

```
BOOST_GENERIC_AUTO_CONCEPT
( (IdentityOf)( (typename) T )
, ( typename type, T )
, ( requires SameType<type, T> )
)
```

Creating Concepts

Starting with the Basics...

```
auto concept IdentityOf<typename T> {
  typename type = T;
  requires SameType<type, T>;
}
```

```
BOOST_GENERIC_AUTO_CONCEPT
( (IdentityOf)( (typename) T )
, ( typename type, T )
, ( requires SameType<type, T> )
)
```


Creating Concepts

Starting with the Basics...

```
auto concept IdentityOf<typename T> {
  typename type = T;
  requires SameType<type, T>;
}
```

```
BOOST_GENERIC_AUTO_CONCEPT
( (IdentityOf)( (typename) T )
, ( typename type, T )
, ( requires SameType<type, T> )
)
```

Creating Concepts

Member Function Requirements

```
auto concept MemberFunctionRequirements<typename T> {
  void T::foo() const;
  T::T( int a, float b );
  T::~T();
}
```


Creating Concepts

Member Function Requirements

```
auto concept MemberFunctionRequirements<typename T> {
  void T::foo() const;
  T::T( int a, float b );
  T::~T();
}
```

```
BOOST_GENERIC_AUTO_CONCEPT
( (MemberFunctionRequirements)( (typename) T )
, ( (void)(this(T) foo)() const )
, ( (this(T))( (int) a, (float) b ) )
, ( (this(T) destroy)() )
)
```


Creating Concepts

Member Function Requirements

```
auto concept MemberFunctionRequirements<typename T> {
  void T::foo() const;
  T::T( int a, float b );
  T::~T();
}
```

```
BOOST_GENERIC_AUTO_CONCEPT
( (MemberFunctionRequirements)( (typename) T )
, ( (void)(this(T) foo)() const )
, ( (this(T))( (int) a, (float) b ) )
, ( (this(T) destroy)() )
)
```

Creating Concepts

Member Function Requirements

```
auto concept MemberFunctionRequirements<typename T> {
  void T::foo() const;
  T::T( int a, float b );
  T::~T();
}
```

```
BOOST_GENERIC_AUTO_CONCEPT
( (MemberFunctionRequirements)( (typename) T )
, ( (void)(this(T) foo)() const )
, ( (this(T))( (int) a, (float) b ) )
, ( (this(T) destroy)() )
)
```


Creating Concepts

Member Function Requirements

```
auto concept MemberFunctionRequirements<typename T> {
  void T::foo() const;
  T::T( int a, float b );
  T::~T();
}
```

```
BOOST_GENERIC_AUTO_CONCEPT
( (MemberFunctionRequirements)( (typename) T )
, ( (void)(this(T) foo)() const )
, ( (this(T))( (int) a, (float) b ) )
, ( (this(T) destroy)() )
)
```

Creating Concepts

Member Function Requirements

```
auto concept MemberFunctionRequirements<typename T> {
  void T::foo() const;
  T::T( int a, float b );
  T::~T();
}
```

```
BOOST_GENERIC_AUTO_CONCEPT
( (MemberFunctionRequirements)( (typename) T )
, ( (void)(this(T) foo)() const )
, ( (this(T))( (int) a, (float) b ) )
, ( (this(T) destroy)() )
)
```


Creating Concepts

Operator Requirements

```
 auto concept HasEqualTo<typename T, typename U> {
  bool operator==(const T& a, const U& b);
}
```


Creating Concepts

Operator Requirements

```
 auto concept HasEqualTo<typename T, typename U> {
  bool operator==(const T& a, const U& b);
}
```

```
BOOST_GENERIC_AUTO_CONCEPT
( (HasEqualTo)( (typename) T, (typename) U )
, ( (bool)(operator equal_to)( (const T&) a, (const U&) b ) )
)
```


Creating Concepts

Operator Requirements

```
 auto concept HasEqualTo<typename T, typename U> {
  bool operator==(const T& a, const U& b);
}
```

```
BOOST_GENERIC_AUTO_CONCEPT
( (HasEqualTo)( (typename) T, (typename) U )
, ( (bool)(operator equal_to)( (const T&) a, (const U&) b ) )
)
```


Creating Concepts

Operator Names

plus +	greater_equal >=	complement ~	minus_assign -=	preincrement ++
minus -	equal_to ==	left_shift <<	multiply_assign *=	postincrement ++
divide /	not_equal_to !=	right_shift >>	divide_assign /=	predecrement --
modulus %	logical_and &&	dereference *	modulus_assign %=	postdecrement --
unary_plus +	logical_or ||	address_of &	bit_and_assign &=	comma ,
negate -	logical_not !	subscript []	bit_or_assign |=	new new
less <	bit_and &	call ()	bit_xor_assign ^=	new_array new []
greater >	bit_or |	assign =	left_shift_assign <<=	delete delete
less_equal <=	bit_xor ^	plus_assign +=	right_shift_assign >>=	delete_array delete []
multiply *	arrow ->	arrow_dereference ->*		

Creating Concepts

Operator Names

plus +	greater_equal >=	complement ~	minus_assign -=	preincrement ++
minus -	equal_to ==	left_shift <<	multiply_assign *=	postincrement ++
divide /	not_equal_to !=	right_shift >>	divide_assign /=	predecrement --
modulus %	logical_and &&	dereference *	modulus_assign %=	postdecrement --
unary_plus +	logical_or ||	address_of &	bit_and_assign &=	comma ,
negate -	logical_not !	subscript []	bit_or_assign |=	new new
less <	bit_and &	call ()	bit_xor_assign ^=	new_array new []
greater >	bit_or |	assign =	left_shift_assign <<=	delete delete
less_equal <=	bit_xor ^	plus_assign +=	right_shift_assign >>=	delete_array delete []
multiply *	arrow ->	arrow_dereference ->*		

Creating Concepts

Conversion Operation Requirements

```
auto concept ExplicitlyConvertible<typename T, typename U> {
  explicit operator U(const T&);
}

auto concept Convertible<typename T, typename U>
  : ExplicitlyConvertible<T, U> {
  operator U(const T&);
}
```


Creating Concepts

Conversion Operation Requirements

```
auto concept ExplicitlyConvertible<typename T, typename U> {
  explicit operator U(const T&);
}

auto concept Convertible<typename T, typename U>
  : ExplicitlyConvertible<T, U> {
  operator U(const T&);
}
```

```
BOOST_GENERIC_AUTO_CONCEPT
( (ExplicitlyConvertible)( (typename) T, (typename) U )
, ( explicit (operator U)( (const T&) ) )
)

BOOST_GENERIC_AUTO_CONCEPT
( (Convertible)( (typename) T, (typename) U )
, ( public ExplicitlyConvertible<T, U > )
, ( (operator U)( (const T&) ) )
)
```


Creating Concepts

More Complicated than It Looks

```
auto concept HasDereference<typename T> {
  typename result_type;
  result_type operator*(T&);
  result_type operator*(T&&);
}
```


Creating Concepts

More Complicated than It Looks

```
auto concept HasDereference<typename T> {
  typename result_type;
  result_type operator*(T&);
  result_type operator*(T&&);
}
```


Creating Concepts

More Complicated than It Looks

```
auto concept HasDereference<typename T> {
  typename result_type;
  result_type operator*(T&);
  result_type operator*(T&&);
}
```

```
BOOST_GENERIC_AUTO_CONCEPT
( (HasDereference)( (typename) T )
, ( typename result_type )
, ( (result_type)(operator dereference)(T&) )
, ( (result_type)(operator dereference)(T&&) )
)
```


Creating Concepts

Concept Value Parameters

```
concept True<bool V> {}
concept_map True<true> {}

auto concept IsEven<intmax_t V>
{
  requires True<V%2==0>;
}
```


Creating Concepts

Concept Value Parameters

```
concept True<bool V> {}
concept_map True<true> {}

auto concept IsEven<intmax_t V>
{
  requires True<V%2==0>;
}
```

```
BOOST_GENERIC_CONCEPT( (True)( (bool) V ) )
…

BOOST_GENERIC_AUTO_CONCEPT
( (IsEven)( (bool) V )
, ( requires True<V%2==0> )
)
```


Creating Concepts

Refinement and Axioms

```
auto concept CopyConstructible<typename T>
  : MoveConstructible<T>, Constructible<T, const T&>
{
  axiom CopyPreservation(T x) {
    T(x) == x;
  }
}
```


Creating Concepts

Refinement and Axioms

```
auto concept CopyConstructible<typename T>
  : MoveConstructible<T>, Constructible<T, const T&>
{
  axiom CopyPreservation(T x) {
    T(x) == x;
  }
}
```

```
BOOST_GENERIC_AUTO_CONCEPT
( (CopyConstructible)( (typename) T )
, ( public MoveConstructible<T>, Constructible1<T, const T&> )
, ( axiom CopyPreservation(T x) {
      T(x) == x;
    }
  )
)
```

Creating Concepts

Everybody's favorite concepts... Iterators (N2914)!!!

```
concept Iterator<typename X> : Semiregular<X> {
  MoveConstructible reference = typename X::reference;
  MoveConstructible postincrement_result;
  requires HasDereference<postincrement_result>;
  reference operator*(X&);
  reference operator*(X&&);
  X& operator++(X&);
  postincrement_result operator++(X&, int);
}
```


Creating Concepts

Everybody's favorite concepts... Iterators (N2914)!!!

```
concept Iterator<typename X> : Semiregular<X> {
  MoveConstructible reference = typename X::reference;
  MoveConstructible postincrement_result;
  requires HasDereference<postincrement_result>;
  reference operator*(X&);
  reference operator*(X&&);
  X& operator++(X&);
  postincrement_result operator++(X&, int);
}
```

```
BOOST_GENERIC_CONCEPT
( (Iterator)( (typename) X ), ( public Semiregular<X> )
, ( (MoveConstructible) reference, typename X::reference )
, ( (MoveConstructible) postincrement_result )
, ( requires HasDereference<postincrement_result> )
, ( (reference)(operator dereference)( (X&) ) )
, ( (reference)(operator dereference)( (X&&) ) )
, ( (X&)(operator preincrement)( (X&) ) )
, ( (postincrement_result)(operator postincrement)( (X&), (int) ) )
)
```


Creating Concepts

Everybody's favorite concepts... Iterators (N2914)!!!

```
concept Iterator<typename X> : Semiregular<X> {
  MoveConstructible reference = typename X::reference;
  MoveConstructible postincrement_result;
  requires HasDereference<postincrement_result>;
  reference operator*(X&);
  reference operator*(X&&);
  X& operator++(X&);
  postincrement_result operator++(X&, int);
}
```

```
BOOST_GENERIC_CONCEPT
( (Iterator)( (typename) X ), ( public Semiregular<X> )
, ( (MoveConstructible) reference, typename X::reference )
, ( (MoveConstructible) postincrement_result )
, ( requires HasDereference<postincrement_result> )
, ( (reference)(operator dereference)( (X&) ) )
, ( (reference)(operator dereference)( (X&&) ) )
, ( (X&)(operator preincrement)( (X&) ) )
, ( (postincrement_result)(operator postincrement)( (X&), (int) ) )
)
```


Creating Concepts

Everybody's favorite concepts... Iterators (N2914)!!!

```
concept Iterator<typename X> : Semiregular<X> {
  MoveConstructible reference = typename X::reference;
  MoveConstructible postincrement_result;
  requires HasDereference<postincrement_result>;
  reference operator*(X&);
  reference operator*(X&&);
  X& operator++(X&);
  postincrement_result operator++(X&, int);
}
```

```
BOOST_GENERIC_CONCEPT
( (Iterator)( (typename) X ), ( public Semiregular<X> )
, ( (MoveConstructible) reference, typename X::reference )
, ( (MoveConstructible) postincrement_result )
, ( requires HasDereference<postincrement_result> )
, ( (reference)(operator dereference)( (X&) ) )
, ( (reference)(operator dereference)( (X&&) ) )
, ( (X&)(operator preincrement)( (X&) ) )
, ( (postincrement_result)(operator postincrement)( (X&), (int) ) )
)
```


Creating Concepts

Everybody's favorite concepts... Iterators (N2914)!!!

```
concept Iterator<typename X> : Semiregular<X> {
  MoveConstructible reference = typename X::reference;
  MoveConstructible postincrement_result;
  requires HasDereference<postincrement_result>;
  reference operator*(X&);
  reference operator*(X&&);
  X& operator++(X&);
  postincrement_result operator++(X&, int);
}
```

```
BOOST_GENERIC_CONCEPT
( (Iterator)( (typename) X ), ( public Semiregular<X> )
, ( (MoveConstructible) reference, typename X::reference )
, ( (MoveConstructible) postincrement_result )
, ( requires HasDereference<postincrement_result> )
, ( (reference)(operator dereference)( (X&) ) )
, ( (reference)(operator dereference)( (X&&) ) )
, ( (X&)(operator preincrement)( (X&) ) )
, ( (postincrement_result)(operator postincrement)( (X&), (int) ) )
)
```


Creating Concepts

Everybody's favorite concepts... Iterators (N2914)!!!

```
concept Iterator<typename X> : Semiregular<X> {
  MoveConstructible reference = typename X::reference;
  MoveConstructible postincrement_result;
  requires HasDereference<postincrement_result>;
  reference operator*(X&);
  reference operator*(X&&);
  X& operator++(X&);
  postincrement_result operator++(X&, int);
}
```

```
BOOST_GENERIC_CONCEPT
( (Iterator)( (typename) X ), ( public Semiregular<X> )
, ( (MoveConstructible) reference, typename X::reference )
, ( (MoveConstructible) postincrement_result )
, ( requires HasDereference<postincrement_result> )
, ( (reference)(operator dereference)( (X&) ) )
, ( (reference)(operator dereference)( (X&&) ) )
, ( (X&)(operator preincrement)( (X&) ) )
, ( (postincrement_result)(operator postincrement)( (X&), (int) ) )
)
```


Creating Concepts

Everybody's favorite concepts... Iterators (N2914)!!!

```
concept Iterator<typename X> : Semiregular<X> {
  MoveConstructible reference = typename X::reference;
  MoveConstructible postincrement_result;
  requires HasDereference<postincrement_result>;
  reference operator*(X&);
  reference operator*(X&&);
  X& operator++(X&);
  postincrement_result operator++(X&, int);
}
```

```
BOOST_GENERIC_CONCEPT
( (Iterator)( (typename) X ), ( public Semiregular<X> )
, ( (MoveConstructible) reference, typename X::reference )
, ( (MoveConstructible) postincrement_result )
, ( requires HasDereference<postincrement_result> )
, ( (reference)(operator dereference)( (X&) ) )
, ( (reference)(operator dereference)( (X&&) ) )
, ( (X&)(operator preincrement)( (X&) ) )
, ( (postincrement_result)(operator postincrement)( (X&), (int) ) )
)
```

Creating Concepts

Everybody's favorite concepts... Iterators (N2914)!!!

```
concept Iterator<typename X> : Semiregular<X> {
  MoveConstructible reference = typename X::reference;
  MoveConstructible postincrement_result;
  requires HasDereference<postincrement_result>;
  reference operator*(X&);
  reference operator*(X&&);
  X& operator++(X&);
  postincrement_result operator++(X&, int);
}
```

```
BOOST_GENERIC_CONCEPT
( (Iterator)( (typename) X ), ( public Semiregular<X> )
, ( (MoveConstructible) reference, typename X::reference )
, ( (MoveConstructible) postincrement_result )
, ( requires HasDereference<postincrement_result> )
, ( (reference)(operator dereference)( (X&) ) )
, ( (reference)(operator dereference)( (X&&) ) )
, ( (X&)(operator preincrement)( (X&) ) )
, ( (postincrement_result)(operator postincrement)( (X&), (int) ) )
)
```


Creating Concepts

Everybody's favorite concepts... Iterators (N2914)!!!

```
concept Iterator<typename X> : Semiregular<X> {
  MoveConstructible reference = typename X::reference;
  MoveConstructible postincrement_result;
  requires HasDereference<postincrement_result>;
  reference operator*(X&);
  reference operator*(X&&);
  X& operator++(X&);
  postincrement_result operator++(X&, int);
}
```

```
BOOST_GENERIC_CONCEPT
( (Iterator)( (typename) X ), ( public Semiregular<X> )
, ( (MoveConstructible) reference, typename X::reference )
, ( (MoveConstructible) postincrement_result )
, ( requires HasDereference<postincrement_result> )
, ( (reference)(operator dereference)( (X&) ) )
, ( (reference)(operator dereference)( (X&&) ) )
, ( (X&)(operator preincrement)( (X&) ) )
, ( (postincrement_result)(operator postincrement)( (X&), (int) ) )
)
```


Creating Concepts

Everybody's favorite concepts... Iterators (N2914)!!!

```
concept Iterator<typename X> : Semiregular<X> {
  MoveConstructible reference = typename X::reference;
  MoveConstructible postincrement_result;
  requires HasDereference<postincrement_result>;
  reference operator*(X&);
  reference operator*(X&&);
  X& operator++(X&);
  postincrement_result operator++(X&, int);
}
```

```
BOOST_GENERIC_CONCEPT
( (Iterator)( (typename) X ), ( public Semiregular<X> )
, ( (MoveConstructible) reference, typename X::reference )
, ( (MoveConstructible) postincrement_result )
, ( requires HasDereference<postincrement_result> )
, ( (reference)(operator dereference)( (X&) ) )
, ( (reference)(operator dereference)( (X&&) ) )
, ( (X&)(operator preincrement)( (X&) ) )
, ( (postincrement_result)(operator postincrement)( (X&), (int) ) )
)
```

Creating Concepts

Everybody's favorite concepts... Iterators (N2914)!!!

```
concept Iterator<typename X> : Semiregular<X> {
  MoveConstructible reference = typename X::reference;
  MoveConstructible postincrement_result;
  requires HasDereference<postincrement_result>;
  reference operator*(X&);
  reference operator*(X&&);
  X& operator++(X&);
  postincrement_result operator++(X&, int);
}
```

```
BOOST_GENERIC_CONCEPT
( (Iterator)( (typename) X ), ( public Semiregular<X> )
, ( (MoveConstructible) reference, typename X::reference )
, ( (MoveConstructible) postincrement_result )
, ( requires HasDereference<postincrement_result> )
, ( (reference)(operator dereference)( (X&) ) )
, ( (reference)(operator dereference)( (X&&) ) )
, ( (X&)(operator preincrement)( (X&) ) )
, ( (postincrement_result)(operator postincrement)( (X&), (int) ) )
)
```


Creating Concepts

Skip ahead to RandomAccessIterators...

```
concept RandomAccessIterator<typename X> : BidirectionalIterator<X>, LessThanComparable<X> {
  MoveConstructible subscript_reference;
  requires Convertible<subscript_reference, const value_type&>;
  X& operator+=(X&, difference_type);
  X operator+ (const X& x, difference_type n) { X tmp(x); tmp += n; return tmp; }
  X operator+ (difference_type n, const X& x) { X tmp(x); tmp += n; return tmp; }
  X& operator-=(X&, difference_type);
  X operator- (const X& x, difference_type n) { X tmp(x); tmp -= n; return tmp; }
  difference_type operator-(const X&, const X&);
  subscript_reference operator[](const X& x, difference_type n);
}
```

I hope you like parentheses.

Creating Concepts

Skip ahead to RandomAccessIterators...

```
concept RandomAccessIterator<typename X> : BidirectionalIterator<X>, LessThanComparable<X> {
  MoveConstructible subscript_reference;
  requires Convertible<subscript_reference, const value_type&>;
  X& operator+=(X&, difference_type);
  X operator+ (const X& x, difference_type n) { X tmp(x); tmp += n; return tmp; }
  X operator+ (difference_type n, const X& x) { X tmp(x); tmp += n; return tmp; }
  X& operator-=(X&, difference_type);
  X operator- (const X& x, difference_type n) { X tmp(x); tmp -= n; return tmp; }
  difference_type operator-(const X&, const X&);
  subscript_reference operator[](const X& x, difference_type n);
}
```

```
BOOST_GENERIC_CONCEPT
( (RandomAccessIterator)( (typename) X ), ( public BidirectionalIterator<X>, LessThanComparable<X> )
, ( (MoveConstructible) subscript_reference )
, ( requires Convertible<subscript_reference, const typename BidirectionalIterator<X>::value_type&> )
, ( (X&)(operator plus_assign)( (X&), (typename BidirectionalIterator<X>::difference_type ) ) )
, ( (X)(operator plus)( (const X&) x, (typename BidirectionalIterator<X>::difference_type) n ) )
, ( (X)(operator plus)( (typename BidirectionalIterator<X>::difference_type) n, (const X&) x ) )
, ( (X&)(operator minus_assign)( (X&), (typename BidirectionalIterator<X>::difference_type) ) )
, ( (X)(operator minus)( (const X&) x, (typename BidirectionalIterator<X>::difference_type) n ) )
, ( (difference_type)(operator minus)( (const X&), (const X&) ) )
, ( (subscript_reference)(operator subscript)( (const X&), (typename BidirectionalIterator<X>::difference_type) ) )
)
```


Creating Concepts

Skip ahead to RandomAccessIterators...

```
concept RandomAccessIterator<typename X> : BidirectionalIterator<X>, LessThanComparable<X> {
  MoveConstructible subscript_reference;
  requires Convertible<subscript_reference, const value_type&>;
  X& operator+=(X&, difference_type);
  X operator+ (const X& x, difference_type n) { X tmp(x); tmp += n; return tmp; }
  X operator+ (difference_type n, const X& x) { X tmp(x); tmp += n; return tmp; }
  X& operator-=(X&, difference_type);
  X operator- (const X& x, difference_type n) { X tmp(x); tmp -= n; return tmp; }
  difference_type operator-(const X&, const X&);
  subscript_reference operator[](const X& x, difference_type n);
}
```

```
BOOST_GENERIC_CONCEPT
( (RandomAccessIterator)( (typename) X ), ( public BidirectionalIterator<X>, LessThanComparable<X> )
, ( (MoveConstructible) subscript_reference )
, ( requires Convertible<subscript_reference, const typename BidirectionalIterator<X>::value_type&> )
, ( (X&)(operator plus_assign)( (X&), (typename BidirectionalIterator<X>::difference_type ) ) )
, ( (X)(operator plus)( (const X&) x, (typename BidirectionalIterator<X>::difference_type) n ) )
, ( (X)(operator plus)( (typename BidirectionalIterator<X>::difference_type) n, (const X&) x ) )
, ( (X&)(operator minus_assign)( (X&), (typename BidirectionalIterator<X>::difference_type) ) )
, ( (X)(operator minus)( (const X&) x, (typename BidirectionalIterator<X>::difference_type) n ) )
, ( (difference_type)(operator minus)( (const X&), (const X&) ) )
, ( (subscript_reference)(operator subscript)( (const X&), (typename BidirectionalIterator<X>::difference_type) ) )
)
```


Creating Concepts

Skip ahead to RandomAccessIterators...

```
concept RandomAccessIterator<typename X> : BidirectionalIterator<X>, LessThanComparable<X> {
  MoveConstructible subscript_reference;
  requires Convertible<subscript_reference, const value_type&>;
  X& operator+=(X&, difference_type);
  X operator+ (const X& x, difference_type n) ~~{ X tmp(x); tmp += n; return tmp; }~~
  X operator+ (difference_type n, const X& x) ~~{ X tmp(x); tmp += n; return tmp; }~~
  X& operator-=(X&, difference_type);
  X operator- (const X& x, difference_type n) ~~{ X tmp(x); tmp -= n; return tmp; }~~
  difference_type operator-(const X&, const X&);
  subscript_reference operator[](const X& x, difference_type n);
}
```

```
BOOST_GENERIC_CONCEPT
( (RandomAccessIterator)( (typename) X ), ( public BidirectionalIterator<X>, LessThanComparable<X> )
, ( (MoveConstructible) subscript_reference )
, ( requires Convertible<subscript_reference, const typename BidirectionalIterator<X>::value_type&> )
, ( (X&)(operator plus_assign)( (X&), (typename BidirectionalIterator<X>::difference_type ) ) )
, ( (X)(operator plus)( (const X&) x, (typename BidirectionalIterator<X>::difference_type) n ) )
, ( (X)(operator plus)( (typename BidirectionalIterator<X>::difference_type) n, (const X&) x ) )
, ( (X&)(operator minus_assign)( (X&), (typename BidirectionalIterator<X>::difference_type) ) )
, ( (X)(operator minus)( (const X&) x, (typename BidirectionalIterator<X>::difference_type) n ) )
, ( (difference_type)(operator minus)( (const X&), (const X&) ) )
, ( (subscript_reference)(operator subscript)( (const X&), (typename BidirectionalIterator<X>::difference_type) ) )
)
```


Creating Concept Maps

The Hard Part Is Over

Creating Concept Maps

Pointers as RandomAccessIterators

```
template<ObjectType T> concept_map RandomAccessIterator<T*> {
  typedef T value_type;
  typedef ptrdiff_t difference_type;
  typedef T& reference;
  typedef T* pointer;
}
```


Creating Concept Maps

Pointers as RandomAccessIterators

```
template<ObjectType T> concept_map RandomAccessIterator<T*> {
  typedef T value_type;
  typedef ptrdiff_t difference_type;
  typedef T& reference;
  typedef T* pointer;
}
```

```
BOOST_GENERIC_CONCEPT_MAP
( ( template ( (class) T ) ), (RandomAccessIterator)(T*)
, ( typedef T value_type )
, ( typedef ptrdiff_t difference_type )
, ( typedef T& reference )
, ( typedef T* pointer )
)
```


Creating Concept Maps

Pointers as RandomAccessIterators

```
template<ObjectType T> concept_map RandomAccessIterator<T*> {
  typedef T value_type;
  typedef ptrdiff_t difference_type;
  typedef T& reference;
  typedef T* pointer;
}
```

```
BOOST_GENERIC_CONCEPT_MAP
( ( template ( (class) T ) ), (RandomAccessIterator)(T*)
, ( typedef T value_type )
, ( typedef ptrdiff_t difference_type )
, ( typedef T& reference )
, ( typedef T* pointer )
)
```

Future Direction

- Automatically Generate Archetypes
- Improve Preprocessor Error Detection
- Create Quickbook Documentation
- Start Testing on Clang
- Add Type-Template Requirements
- Parse Variadic Concepts
- Allow Ref-Qualifiers on Member-Function Requirements
- Finish Implementing the Concepts of N2914
- Do Basic Syntax Checking on Axioms
- Optimize Preprocessing
- Make a Backend Targeting ConceptGCC and ConceptClang
- Get Boost.Generic Reviewed and [Hopefully] into Boost
- Create Concepts for BGL, Boost.GIL and other Boost Libraries
- Automatic Creation of Type Erasure Constructs (I.e. Boost.Any, Boost.Function)

Questions

