C++ Language Support for Generic Programming

Jeremy Siek, Douglas Gregor, Ronald Garcia, Jeremiah Willcock, Jaakko Järvi, and Andrew Lumsdaine
What is Generic Programming?

- Paradigm for software development
- Express algorithms (and data structures) in terms of abstract requirements
  - Applicable to many data types
  - Runtime performance of concrete implementations
  - Alternative implementations for special-case data structures
- Distinct from template metaprogramming
Outline

- Introduction to GP extensions for C++
  - Generic functions & types
  - Concepts
  - Models
- Existing practice & the Standard Library
- Impact on the C++ community
Introduction to Generic Programming Extensions for C++
Vector equality

- **std::vector** has this equality operator:
  
  ```cpp
template<typename T, typename Alloc>
  bool operator==(const vector<T, Alloc>& x, const vector<T, Alloc>& y);
  ```

- When can we instantiate this function?
  - Need to be able to compare T objects.
  - This requirement is **implicit** in the source code
Explicit requirements

- Make the requirement explicit:
  
  ```cpp
  template<typename T, typename Alloc>
  where { EqualityComparable<T> }
  bool operator==(const vector<T, Alloc>& x, const vector<T, Alloc>& y);
  ```

- where clause states requirements
  - EqualityComparable is a concept
  - “T must be EqualityComparable”
Equality Comparable concept

template<
    typeid T>
concept EqualityComparable
{
    bool operator==(const T&, const T&);
    bool operator!=(const T&, const T&);

    // == is an equivalence relation
    // != is the complement of ==
};

- A concept gives a name to a set of requirements
  - Syntax
  - Semantics
- A type models the concept if it satisfies the requirements
Type-checking calls

- We can now type-check calls to:
  
  ```
  template<typename T, typename Alloc>
  where { EqualityComparable<T> }
  bool operator==(const vector<T, Alloc>& x, const vector<T, Alloc>& y);
  ```

  ```
  vector<int> iv1, iv2;
  if (iv1 == iv2) { ... }
  vector<my_type> mv1, mv2;
  if (mv1 == mv2) { ... }
  ```

- We need to ensure that `int` and `my_type` model `EqualityComparable`
Models of EqualityComparable

- int models EqualityComparable
  - The Standard Library asserts this
- Does my_type model EqualityComparable?
  - Only if syntax and semantics match
  - If yes, write a model definition:

    ```
    model EqualityComparable<my_type> { };
    ```
Opaque template parameters

- Let’s add the implementation:

  ```cpp
template<
typeid T, typeid Alloc>
  where { EqualityComparable<T> }
  bool operator==(const vector<T, Alloc>& x,
                 const vector<T, Alloc>& y)
  {
    return x.size() == y.size()
       && equal(x.begin(), x.end(), y.begin());
  }
```

- `typename` eliminates type checking: you can do anything to `typename` types
- `typeid` enables type checking: you can do nothing but what you require
- Two-phase type checking: like C++ already has!
What about the call to \texttt{equal}?

- \texttt{equal} has this signature with concepts:
  
  \begin{verbatim}
  template<typename Iter1, typename Iter2>
  where { EqualityComparable2<
    InputIterator<Iter1>::value_type,
    InputIterator<Iter2>::value_type> } 

  bool
  equal(Iter1 first1, Iter1 last1, Iter2 first2);
  \end{verbatim}

- \texttt{EqualityComparable2} is a new concept:
  
  \begin{verbatim}
  template<typename T, typename U>
  concept EqualityComparable2
  {
    bool operator==(const T& , const U&);
    bool operator!=(const T& , const U&);
  };
  \end{verbatim}
What about the call to `equal`?

- `equal` has this signature with concepts:
  ```
  template<
typeid Iter1, typeid Iter2>
  where { EqualityComparable2<
    InputIterator<Iter1>::value_type,
    InputIterator<Iter2>::value_type> }

  bool
  equal(Iter1 first1, Iter1 last1, Iter2 first2);
  ```

- **Both** `Iter1` and `Iter2` **must** model the `InputIterator` concept.
What about the call to `equal`?

- `equal` has this signature with concepts:
  
  ```
  template<
    typeid Iter1, typeid Iter2>
  where {
    EqualityComparable2<
      InputIterator<Iter1>::value_type,
      InputIterator<Iter2>::value_type>
  }
  ```

- `InputIterator` concept has associated types
  
  - Supersede the use of traits
  - Note: no `typename`!
EqualityComparable vs. EqualityComparable2

- `vector ==` requires `EqualityComparable`
- `std::equal` requires `EqualityComparable2`
- How are they related?
EqualityComparable vs. EqualityComparable2

- vector == requires EqualityComparable
- std::equal requires EqualityComparable2
- How are they related?
  - EqualityComparable2 is more general
  - We call EqualityComparable a refinement of EqualityComparable2.
Refinement

- A concept \( B \) **refines** concept \( A \) if \( B \) includes all of the requirements of \( A \).
  - Akin to inheritance of abstract classes
  - RandomAccessIterator **refines** BidirectionalIterator

- **A better way to define** `EqualityComparable`:
  ```cpp
template<typename T>
concept EqualityComparable : EqualityComparable2<T, T> { };
```
**InputIterator Concept**

```cpp
template<typeid Iter>
concept InputIterator : EqualityComparable<Iter>,
    Assignable<Iter>,
    CopyConstructible<Iter>
{
    typename value_type;
    typename difference_type;
    require Integral<difference_type>;
    const value_type& operator*(const Iter&);
    Iter& operator++(Iter&);
}
```
“Make the hard things possible”

template<typeid X>
concept Sequence : Container<X>
{
    typename value_type;
    typename iterator;
    require ForwardIterator<iterator>,
        ForwardIterator<iterator>::value_type == value_type;

template<typeid Iter>
    where {
        Convertible<InputIterator<Iter>::value_type,
            value_type> }
    X::X(Iter first, Iter last);
};
Summary of features

- Concepts
  - Refinement
  - Associated types
  - Pseudo-signatures

- Explicit models of concepts

- Where clauses describe requirements
  - Concept
  - Same-type

- Concept-based function selection
Existing practice and the Standard Library
Standard Library ↔ Concepts

- Requirements tables ↔ Concepts
- Template type parameter names ↔ Where clauses
- Loose syntactic requirements ↔ Concepts + where clauses
- Traits ↔ Associated types, model declarations
Existing practice

- Excerpt from SGI STL documentation:

  As written with our proposal:

  ```cpp
  template<
typeid X>
  concept DefaultConstructible
  {
      X::X();
  };

  model DefaultConstructible<int> { };  
  model DefaultConstructible<vector<double>> { };

  template<
typename T>
  model DefaultConstructible<vector<T>> { };
  ```

- Excerpt from SGI STL documentation:

  As written with our proposal:

  ```cpp
  template<
typeid X>
  concept DefaultConstructible
  {
      X::X();
  };

  model DefaultConstructible<int> { };  
  model DefaultConstructible<vector<double>> { };

  template<
typename T>
  model DefaultConstructible<vector<T>> { };
  ```

Default Constructible

- Utilities
  Category: utilities
  Component type: concept

- Description
  A type is DefaultConstructible if it has a default constructor, that is, if it is possible to construct an object of that type without initializing the object to any particular value.

- Refinement of
  Associated types
  Notation
  \* A type that is a model of DefaultConstructible
  \* An object of type \( X \)

- Definitions
  Valid expressions

<table>
<thead>
<tr>
<th>Name</th>
<th>Expression</th>
<th>Type requirements</th>
<th>Return type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default constructor</td>
<td>( X() )</td>
<td></td>
<td>( X )</td>
</tr>
<tr>
<td>Default constructor</td>
<td>( X; )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Expression semantics

<table>
<thead>
<tr>
<th>Name</th>
<th>Expression</th>
<th>Precondition</th>
<th>Semantics</th>
<th>Postcondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default constructor</td>
<td>( X() )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Default constructor</td>
<td>( X; )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Complexity guarantees

- Models
  - int
  - `vector<double>`
[20.1.3/1] In the following Table 30, $T$ is a type to be supplied by a C++ program instantiating a template, $t$ is a value of type $T$, and $u$ is a value of type const $T$.

Table 30: CopyConstructible requirements

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(t)$</td>
<td>$t$</td>
<td>is equivalent to $T(t)$</td>
</tr>
<tr>
<td>$T(u)$</td>
<td>$u$</td>
<td>is equivalent to $T(u)$</td>
</tr>
<tr>
<td>$t.\sim T()$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$&amp;t$</td>
<td>$T*$</td>
<td>denotes the address of $t$</td>
</tr>
<tr>
<td>$&amp;u$</td>
<td>const $T*$</td>
<td>denotes the address of $u$</td>
</tr>
</tbody>
</table>

```
template<typeid T>
concept CopyConstructible
{
    T::T(T&);
    T::T(const T&);
    T::~T();
    T* operator&(T&);
    const T* operator&(const T&);
};
```
Impact on the C++ community
Impact on users

- Vast majority of code still works, unchanged
- Generic Programming becomes easier
  - Stronger typing for generic functions
  - Clean, concise error messages
  - Replace learning template tricks with learning Generic Programming
- Application code will need explicit model definitions
  - Not very often: the Standard Library handles most of them
  - Similar effort to inheriting abstract base classes
Impact on Standard Libraries

- Add `where` clauses as specified by the standard
- Option: conditionally enable `where` clauses for a single C++03/C++0x library
  - Tag dispatching & associated types a little harder
- Option: leave template parameters as `typename`
  - Checks uses of templates
  - ... but not definitions!
  - Allows clever optimizations in library implementations
Impact on compilers

- Several parts to implementation
  - Concepts, refinement, pseudo-signatures similar to class templates
  - Models similar to class template specializations
  - Type-checking is still two-phase lookup
  - Where clauses are quite simple

- We’re addressing this on several fronts
  - Nontrivial portion of STL implemented in *G*
  - Prototype in GCC
Summary of the proposal

- We propose complete support for Generic Programming in C++ that:
  - Makes Generic Programming accessible
  - Reflects existing practice
  - Is expressive enough for the entire C++ Standard Library
  - Is implementable in current C++ compilers
Explicit vs. Implicit Model Declarations

Jeremy Siek, Douglas Gregor, Ronald Garcia, Jeremiah Willcock, Jaakko Järvi, and Andrew Lumsdaine
Definitions Recap

- A **model declaration** states that a type or set of types meet the syntactic and semantic requirements of a concept.

- With **implicit** model declarations, the compiler performs **structural matches** to determine if a set of types model a concept.

- With **explicit** model declarations, the **user states** that a set of types model a concept.
# Implicit Model Declarations

**Benefits**
- User need not fully understand concepts
- Backward compatibility
- Matches what we do now in C++ (sometimes [*])

**Problems**
- Accidental conformance [*]
- Implementation complexity [*]

[*] Indicates that we have examples for these points.
Accidental Conformance

- Occurs when the semantics assumed due to a structural match are incorrect.
- Consider `istream_iterator`:
  - Structurally matches `ForwardIterator`.
  - Semantically matches `InputIterator`.
  - With implicit models, compiler can’t catch this error.
- `vector<int> v(istream_iterator<int>(cin), istream_iterator<int>())`;
Accidental Conformance

- Occurs when the semantics assumed due to a structural match are incorrect.

- Consider `istream_iterator`:
  - Structurally matches `ForwardIterator`.
  - Semantically matches `InputIterator`.
  - With implicit models, compiler can’t catch this error.

- `vector<int> v(istream_iterator<int>(cin), istream_iterator<int>());`

- Why doesn’t this problem happen now?
Accidental Conformance

- Occurs when the semantics assumed due to a structural match are incorrect.

- Consider `istream_iterator`:
  - Structurally matches `ForwardIterator`.
  - Semantically matches `InputIterator`.
  - With implicit models, compiler can’t catch this error.

- `vector<int> v(istream_iterator<int>(cin), istream_iterator<int>())`;

- Why doesn’t this problem happen now?
  - `iterator_category` is an explicit model declaration!
Nominal conformance in N1782

- The refinement hierarchy uses nominal conformance
- Overloading is not structural (see Section 6.3 of N1782)
Implementation Complexity

- Structural matching requires SFINAE-like behavior
  - For arbitrary types & expressions
  - Compiler must quietly “back out” if structural match fails
  - Compiler initiates search for structural matches
- Implementors have claimed this is difficult
# Explicit Model Declarations

<table>
<thead>
<tr>
<th>Benefits</th>
<th>Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>□ Strong semantic guarantees</td>
<td>□ Not fully backward-compatible [*]</td>
</tr>
<tr>
<td>■ No accidental conformance</td>
<td>□ Users must understand concepts [*]</td>
</tr>
<tr>
<td>■ Better optimization [*]</td>
<td>□ Users must write model declarations</td>
</tr>
<tr>
<td>□ Matches what we do now in C++ (sometimes)</td>
<td></td>
</tr>
</tbody>
</table>

[*] Indicates that we have examples for these points.
Backward Compatibility

- User will need to add model declarations
- Some ways to avoid these:
  - Where model declarations already exist as traits, we can create model templates (e.g., in the standard library)
  - Falling back to (weak) structural matching can be accomplished with model templates and metaprogramming.
  - Some concepts (e.g., DefaultConstructible) may be so basic that the compiler should add the models.
void example(const std::vector<int>& v) {
    for (auto& i : v) {
        std::cout << i << " ";
    }
    std::cout << std::endl;
}
Better Optimization

- Explicit models are semantic guarantees
  - Needed by generic functions
  - Usable by compilers & optimizers

- Examples:
  - CopyConstructible/Assignable copy propagation
  - Parallelize RandomAccessIterator loops
  - VectorSpace loop fusion

- Is this feasible?
  - It’s still somewhat of a research topic, but…
  - Could find some important optimization opportunities
Users must understand concepts

- What if the user forgets a model declaration?
  - Compiler provides message like “type Foo does not model the InputIterator concept.”
  - User needs to know what that means.

- Mitigating factor: the compiler can suggest model declarations.

    sort_list.cpp:7: error: no matching function for call to 'sort(std::_List_iterator<int>, std::_List_iterator<int>)'
    /.../bits/stl_algo.h:2559: note: candidates are: void std::sort(_RandomAccessIterator,
    _RandomAccessIterator) [with _RandomAccessIterator = std::_List_iterator<int>]
    <failed requirements>
    sort_list.cpp:7: note: unable to locate a model
    'std::MutableRandomAccessIterator<std::_List_iterator<int> >'
    sort_list.cpp:7: note: for concept requirement
    'std::MutableRandomAccessIterator<_RandomAccessIterator>' (you may need to write a model definition)
Our View

- Concepts have semantic requirements
  - We need users to state that their types meet these requirements
  - It’s common practice to do so (e.g., iterators)
  - Optimization, simpler implementation just side benefits

- Backward compatibility is one-time hit
  - Big benefits once the jump is made
  - Compilers, libraries, and tools can help bridge the gap.
Pseudo-signatures vs. Usage Patterns

Jeremy Siek, Douglas Gregor, Ronald Garcia, Jeremiah Willcock, Jaakko Järvi, and Andrew Lumsdaine
Definitions recap

- **Pseudo-signatures** look like function declarations or definitions, but match a class of functions.

- **Usage patterns** illustrate the syntax that generic functions will use, but match a class of functions.
Semantics are the same

- One-sided leniency for both syntax kinds
  - Strict checking of concept uses in generic functions
  - “Convertible-to” okay in models (implicit and explicit)
  - Misunderstandings and prior issues obscure this

- Can both express concepts in the Standard Library
  - -> operator gives usage patterns some trouble
  - OutputIterator is hard on both syntax kinds
Pseudo-signatures

- Declarations look like function declarations:

  ```
  bool operator==(const T&, const U&);
  bool operator!=(const T&, const U&);
  
  template<
typeid Iter>
  where {
    Convertible<InputIterator<Iter>::value_type,
    value_type> }
  X::X(Iter first, Iter last);
  ```
Usage patterns

- Declare variables to be used in expressions:
  ```
  T t;
  U u;
  Input_Iterator Iter;
  Iter i, j;
  static_assert
    Convertible<Iter::value_type, value_type> {};
  ```

- Write expressions describing how objects can be used:
  ```
  (bool)(t == u);
  (bool)(t != u);
  X(i, j);
  ```
operator->

- Not possible in general for usage patterns (see N1782)
- Pseudo-signatures use, e.g.,:
  
  ```cpp
  T* operator->(const X&);
  ```

- Impact: Probably need a built-in Arrow concept to support usage patterns
**Compound expressions**

- **Usage patterns can represent compound expressions:**
  ```
  T a, b, c;
  (T)(a * b + c);
  ```
  - Internal type of \( a \times b \) is unknown/irrelevant

- **Pseudo-signatures require naming the type:**
  ```
  typename mul_type;
  mul_type operator*(T, T);
  T operator+(mul_type, T);
  ```
  - \( \text{mul\_type} \) can use a `decltype` default to save the user some effort
Compound expressions: Impact

- Input and output iterator postincrement is specified as a compound expression
  - Just list the expression with usage patterns
  - Requires introduction of a hidden type for each concept
    - See N1758 for details
Should models look like concepts?

- If a model looks like one of these:

```cpp
static_assert FG<X> { 
    void f(X& a) { a.f(); } 
    void X::g() { g(*this); } 
};
```

```cpp
model FG<X> { 
    void f(X& a) { a.f(); } 
    void X::g() { g(*this); } 
};
```

- What should the concept look like?

```cpp
concept FG<class T> { 
    T a; 
    f(a); 
    a.g(); 
};
```

```cpp
template<typeid T>
concept FG { 
    void f(X& a); 
    void X::g(); 
};
```
Expression templates

- Expression templates require lots of “hidden” types
  - One for each subexpression

- Impact:
  - Usage patterns: list every expression in the function body as a usage pattern
  - Pseudo-signatures: add pseudo-types for each subexpression

- Distinction between “opaque” and “non-opaque” types in N1758 offers one solution.
Syntax: Pseudo-signatures

- **Pros:**
  - Match free & member function syntax
  - Model syntax matches concept syntax
  - Concepts resemble abstract classes
  - Very little new parser technology

- **Cons:**
  - More verbose than usage patterns
  - Compound expressions are more painful
  - Standard Library (and other generic libraries) use usage patterns/valid expressions
Syntax: Usage patterns

- **Pros:**
  - Concise
  - Similar to existing requirements tables
  - Very little new parser technology

- **Cons:**
  - Usage patterns look different from the things they describe (no cut ‘n’ paste coding)
  - Need built-in Arrow concept
  - Determining which declarations declare values vs. types can be confusing.
Our View

- It’s just a syntax issue
- Which is “more readable”?
  - It’s a toss up: good and bad examples for both
  - We like how pseudo-signatures look like the function declarations they match
    - How does one implement a new model of a concept? Copy ‘n’ paste!
    - No “hidden” template requirements
Associated Types

Jeremy Siek, Douglas Gregor, Ronald Garcia, Jeremiah Willcock, Jaakko Järvi, and Andrew Lumsdaine
Where do associated types live?

- N1758: As associated types of concepts
  - `ForwardIterator<Iter>::value_type`

- N1782: As member types of types involved in concepts
  - `Iter::value_type`
Ambiguity with member types

```cpp
concept Callable1<F,T1> {
    typename F::result_type;
    F f; T1 t1;
    (F::result_type)(f(t1));
};

struct negate {
    template<typename T> operator()(T x) { return -x; }
};

static_assert Callable1<negate, int> {
    typedef int negate::result_type;
};

static_assert Callable1<negate, float> {
    typedef float negate::result_type;
};

// (what is negate::result_type?)
```
Our View

- Using member types can be awkward
  - Potential ambiguities with syntax
  - Odd to add member types to non-classes
  - There isn’t always a “main type” to hang on

- Associated types are naturally part of concepts
  - Multiple types of a concept participate to produce associated types
  - Traits use this same level of indirection
Semi-structured additional slides

Jeremy Siek, Douglas Gregor, Ronald Garcia, Jeremiah Willcock, Jaakko Järvi, and Andrew Lumsdaine
std::distance with concepts

template<typeid Iter> where { InputIterator<Iter> }
InputIterator<Iter>::difference_type
distance(Iter first, Iter last)
{
    InputIterator<T>::difference_type result = 0;
    for (; first != last; ++first)
        ++result;
    return result;
}

- Things to notice:
  - We’ve added more to the declaration
  - The body really hasn’t changed
std::distance with concepts

template<typeid Iter> where { InputIterator<Iter> } 
InputIterator<Iter>::difference_type
distance(Iter first, Iter last)
{
    InputIterator<T>::difference_type result = 0;
    for (; first != last; ++first)
        ++result;
    return result;
}
template<typeid Iter>
    where { RandomAccessIterator<Iter> }
RandomAccessIterator<Iter>::difference_type
distance(Iter first, Iter last)
{
    return last - first;
}
A concept is a set of requirements

- Syntactic: functions, operators, types
- Semantic: what functions do, function complexity
- Like requirements tables in the standard
InputIterator Concept

template<
typeid Iter>
concept InputIterator : EqualityComparable<Iter>,
Assignable<Iter>,
CopyConstructible<Iter>
{
  typename value_type;
  typename difference_type;

  const value_type& operator*(const Iter&);
  Iter& operator++(Iter&);
};

- Things to note:
  - Has types `value_type` and `difference_type`
  - Has pseudo-signatures for operators `++` and `*`
  - Refines `EqualityComparable,Assignable, and CopyConstructible`
Opaque template parameters

```cpp
template<
typeid  Iter> where { InputIterator<Iter> }
InputIterator<Iter>::difference_type
distance(Iter first, Iter last)
{
    InputIterator<T>::difference_type result = 0;
    for (; first != last; ++first)
        ++result;
    return result;
}
```

- `typename` eliminates type checking: you can do anything to `typename` types.
- `typeid` provides type checking: you can do nothing but what you require
- Type checking has two phases already!
where clauses

template<typeid Iter> where { InputIterator<Iter> } InputIterator<Iter>::difference_type distance(Iter first, Iter last) {
    InputIterator<Iter>::difference_type result = 0;
    for (; first != last; ++first)
        ++result;
    return result;
}

where clauses constrain templates

- Example: Iter must model the InputIterator concept
- User must pass in types that meet these requirements
- Implementor must only use types and operations mandated by the requirements
Associated types

template<
typeid Iter> where {
  Iterator<Iter> 
}
Iterator<Iter>::difference_type
distance(Iter first, Iter last)
{
  Iterator<T>::difference_type result = 0;
  for (; first != last; ++first)
    ++result;
  return result;
}

- Associated types are additional types used to specify a concept
  - Remember difference_type from Iterator?
  - Supersedes traits
  - Note: no typename!
How do we call distance?

- Just like we always have:
  ```cpp
  list<int> l; // initialize l
  cout << "Length = "
  << distance(l.begin(), l.end()) << endl;
  ```

- Most uses of concepts will be invisible
  - They provide better type safety
  - It’s easier to write correct templates
  - Standard library provides its own models
  - Compiler will provide some concepts “for free.”
More same-type constraints

- Consider `std::merge`:
  ```cpp
template<
typeid Iter1, typeid Iter2, typeid OIter,
typeid T>
where {
  InputIterator<Iter1>::value_type == T,
  InputIterator<Iter2>::value_type == T,
 Convertible<OutputIterator<OIter>::value_type, T>,
StrictWeakOrdering<T> }
OIter merge(Iter1 first1, Iter1 last1,
  Iter2 first2, Iter2 last2, OIter out);
```
## Sequence requirements

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>assertion/note pre/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>X(n, t)</td>
<td></td>
<td>post: size() == n</td>
</tr>
<tr>
<td>X a(n, t)</td>
<td></td>
<td>constructs a sequence with n copies of t</td>
</tr>
<tr>
<td>X(i, j)</td>
<td></td>
<td>post: size() == distance between i and j</td>
</tr>
<tr>
<td>X a(i, j)</td>
<td></td>
<td>constructs a sequence equal to the range [i, j)</td>
</tr>
<tr>
<td>a.insert(p, t)</td>
<td>iterator</td>
<td>inserts a copy of t before p</td>
</tr>
<tr>
<td>a.insert(p, n, t)</td>
<td>void</td>
<td>inserts n copies of t before p</td>
</tr>
<tr>
<td>a.insert(p, i, j)</td>
<td>void</td>
<td>pre: i and j are not iterators into a. inserts copies of elements in [i, j) before p</td>
</tr>
<tr>
<td>a.erase(q)</td>
<td>iterator</td>
<td>erases the element pointed to by q</td>
</tr>
<tr>
<td>a.erase(q1, q2)</td>
<td>iterator</td>
<td>erases the elements in the range [q1, q2).</td>
</tr>
<tr>
<td>a.clear()</td>
<td>void</td>
<td>erase(begin(), end())</td>
</tr>
<tr>
<td>a.assign(i, j)</td>
<td>void</td>
<td>post: size() == 0</td>
</tr>
<tr>
<td>a.assign(n, t)</td>
<td>void</td>
<td>pre: t is not a reference into a. Replaces elements in a with a copy of [i, j).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pre: t is not a reference into a. Replaces elements in a with n copies of t.</td>
</tr>
</tbody>
</table>
Why propose language support?

- Generic Programming is gaining acceptance
  - C++ Standard Library is prototypical example
  - Still too much work to use GP
  - Java, C# now support templates/generics
- Conventions are okay, but language support is better
  - The ideas of GP can be expressed more clearly
  - Better tool support
- We are at the tipping point
Revisiting vector ==

- How does this type-check?:

  template<typename T, typename Alloc>
  bool operator==(const vector<T, Alloc>& x, const vector<T, Alloc>& y) {
    return x.size() == y.size()
    && equal(x.begin(), x.end(), y.begin());
  }

- The only operations on T objects are in std::equal
- std::equal requires EqualityComparable2
  - Okay, since EqualityComparable implies EqualityComparable2
- std::equal requires InputIterators
  - Okay, since vector<T>::const_iterator models RandomAccessIterator.
“Small” example

template<typeid T, int N>
concept Small
{
   require sizeof(T) <= N;
};

template<typename T, int N>
   where {sizeof(T) <= N }
model Small<T, N> {};

template<typename T>
   where {Small<T,200> } void f(T&);
template<typename T> void f(T&);

template<typename T>
   void foo(const T& t)
   {
      f(t);
   }