# Concept Checking: Binding Parametric Polymorphism in C++

Jeremy Siek Andrew Lumsdaine
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556
{jsiek,lums}@lsc.nd.edu

**Abstract.** Generic programming in C++ is characterized by the use of template parameters to represent abstract data types (or "concepts"). However, the C++ language itself does not provide a mechanism for explicitly handling concepts. As a result, it can be difficult to insure that a concrete type meets the requirements of the concept it is supposed to represent. Error messages resulting from incorrect use of a concrete type can be particularly difficult to decipher. In this paper we present techniques to check parameters in generic C++ libraries. Our techniques use standard C++ and introduce no run-time overhead.

## 1 Introduction

A *concept* is a set of requirements (valid expressions, associated types, semantic invariants, complexity guarantees, etc.) that a type must fulfill to be correctly used within the context of a generic algorithm. In C++, concepts are represented by formal template parameters to function templates (generic algorithms). However, C++ has no explicit mechanism for representing concepts — template parameters are merely placeholders. By convention, these parameters are given names corresponding to the concept that is required, but a C++ compiler does not enforce compliance to the concept when the template parameter is bound to an actual type.

Naturally, if a generic algorithm is invoked with a type that does not fulfill at least the syntactic requirements of the concept, a compile-time error will occur. However, this error will not *per se* reflect the fact that the type did not meet all of the requirements of the concept. Rather, the error may occur deep inside the instantiation hierarchy at the point where an expression is not valid for the type, or where a presumed associated type is not available. The resulting error messages are largely uninformative and basically impenetrable.

What is required is a mechanism for enforcing "concept safety" at (or close to) the point of instantiation. Re-engineering the template system in C++ to accommodate concepts is a daunting task (and we feel there are better mechanisms than simply adding yet more onto C++ for providing concept-safe genericity). We present instead a tech-

nique that uses standard C++ constructs to enforce early concept compliance and that provides more informative error messages upon non-compliance. We have applied this mechanism to the SGI implementation of STL, and the changes are now in the main distribution.

Note that with this technique we only address the syntactic requirements of concepts (the valid expressions and associated types). We do not address the semantic invariants or complexity guarantees, which are also part of concept requirements..

## 2 Example

We present a simple example to illustrate incorrect usage of a template library and the resulting error messages. In the code below, the generic `std::stable_sort()` algorithm from the Standard Template Library (STL) [1, 6, 7] is applied to a linked list.

```
bad_error_eg.cpp:
 1  #include <list>
 2  #include <algorithm>
 3
 4  struct foo {
 5    bool operator<(const foo&) const { return false; }
 6  };
 7  int main(int, char*[]) {
 8    std::list<foo> v;
 9    std::stable_sort(v.begin(), v.end());
10    return 0;
11  }
```

Here, the `std::stable_sort()` algorithm is prototyped as follows:

```
template <class RandomAccessIterator>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last);
```

Attempting to compile this code with Gnu C++ produces the following compiler error. The output from other compilers is listed in the Appendix.

```
stl_algo.h: In function ‘void __merge_sort_loop<_List_iterator
  <foo,foo &,foo *>, foo *, int>(_List_iterator<foo,foo &,foo *>,
  _List_iterator<foo,foo &,foo *>, foo *, int)’:
stl_algo.h:1448:   instantiated from ‘__merge_sort_with_buffer
  <_List_iterator<foo,foo &,foo *>, foo *, int>(
   _List_iterator<foo,foo &,foo *>, _List_iterator<foo,foo &,foo *>,
   foo *, int *)’
stl_algo.h:1485:   instantiated from ‘__stable_sort_adaptive<
  _List_iterator<foo,foo &,foo *>, foo *, int>(_List_iterator
  <foo,foo &,foo *>, _List_iterator<foo,foo &,foo *>, foo *, int)’
stl_algo.h:1524:   instantiated from here
stl_algo.h:1377: no match for ‘_List_iterator<foo,foo &,foo *> & -
  _List_iterator<foo,foo &,foo *> &’
```

In this case, the fundamental error is that `std:list::iterator` does not model the concept of RandomAccessIterator. The list iterator is only bidirectional, not fully random access (as would be a vector iterator). Unfortunately, there is nothing in the error message to indicate this to the user.

To a C++ programmer having enough experience with template libraries the error is obvious. However, for the uninitiated, there are several reasons why this message would be hard to understand.

- The location of the error, line 9 of `bad_error_eg.cpp` is not pointed to by the error message, despite the fact that Gnu C++ prints up to 4 levels deep in the instantiation stack.
- There is no textual correlation between the error message and the documented requirements for `std::stable_sort()` and for RandomAccessIterator.
- The error message is overly long, listing functions internal to the STL that the user does not (and should not!) know or care about.
- With so many internal library functions listed in the error message, the programmer could easily infer that the error is due to the library, rather than to his or her own code.

The following is an example of what we might expect from a more informative message (and is in fact what the system presented here can produce):

```
concept_checks.h: In method ‘void LessThanComparable_concept
  <_List_iterator<foo,foo &,foo *> >::constraints()’:
concept_checks.h:334:   instantiated from ‘RandomAccessIterator_concept
  <_List_iterator<foo,foo &,foo *> >::constraints()’
bad_error_eg.cpp:9:   instantiated from ‘stable_sort<_List_iterator
  <foo,foo &,foo *> >(_List_iterator<foo,foo &,foo *>,
  _List_iterator<foo,foo &,foo *>)’
concept_checks.h:209: no match for ‘_List_iterator<foo,foo &,foo *> &
  < _List_iterator<foo,foo &,foo *> &’
```

This message rectifies several of the shortcomings of the standard error messages.

- The location of the error, `bad_error_eg.cpp:9` is specified in the error message.
- The message refers explicitly to concepts that the user can look up in the STL documentation (RandomAccessIterator).
- The error message is now much shorter and does not reveal internal STL functions.
- The presence of `concept_checks.h` and `constraints()` in the error message alerts the user to the fact that the error lies in the user code and not in the library implementation.

## 3 Concept Checks

Ideally we would like to catch, and indicate, the concept violation at the point of instantiation. As mentioned in D&E [8], the error can be caught by exercising all of the requirements needed by the function template. Exactly how the requirements (the

valid expressions in particular) are exercised is a tricky issue, since we want the code to be compiled — *but not executed*. Our approach is to exercise the requirements in a separate function that is assigned to a function pointer. In this case, the compiler will instantiate the function but will not actually invoke it. In addition, an optimizing compiler will remove the pointer assignment as "dead code" (though the run-time overhead added by the assignment would be trivial in any case). It might be conceivable for a compiler to skip the semantic analysis and compilation of the constraints function in the first place, which would make our function pointer technique ineffective. However, this is unlikely because removal of unnecessary code and functions is typically done in later stages of a compiler. We have successfully used the function pointer technique with GNU C++, Microsoft Visual C++, and several EDG-based compilers (KAI C++, SGI MIPSpro). The following code shows how this technique can be applied to the `std::stable_sort()` function:

```
template <class RandomAccessIterator>
void stable_sort_constraints(RandomAccessIterator i) {
  typename std::iterator_traits<RandomAccessIterator>
    ::difference_type n;
  i += n;  // exercise the requirements for RandomAccessIterator
  ...
}
template <class RandomAccessIterator>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last) {
  typedef void (*fptr_type)(RandomAccessIterator);
  fptr_type x = &stable_sort_constraints;
  ...
}
```

There is often a large set of requirements that need to be checked, and it would be cumbersome for the library implementor to write constraint functions like `stable_sort_constraints()` for every public function. Instead, we group sets of valid expressions together, according to the definitions of the corresponding concepts. For each concept we define a concept checking class template (using the `_concept` suffix as a naming convention). The template parameter is for the type to be checked. The class contains a `contraints()` member function which exercises all of the valid expressions of the concept. The objects used in the constraints function, such as `n` and `i`, are declared as data members of the concept checking class.

```
template <class Iter>
struct RandomAccessIterator_concept {
  void constraints() {
    i += n;
    ...
  }
  typename std::iterator_traits<RandomAccessIterator>
    ::difference_type n;
  Iter i;
  ...
};
```

We can still use the function pointer mechanism to cause instantiation of the constraints function, however now it will be a member function pointer. To make it easy

for the library implementor to invoke the concept checks, we wrap the member function pointer mechanism in a macro named `REQUIRE`. The following code snippet shows how to use `REQUIRE` to make sure that the iterator is a RandomAccessIterator.

```
template <class RandomAccessIter>
void stable_sort(RandomAccessIter first, RandomAccessIter last)
{
  REQUIRE(RandomAccessIter, RandomAccessIterator);
  ...
}
```

The definition of the `REQUIRE` is as follows. The `type_var` is the type we wish to check, and `concept` is the name that corresponds to the concept checking class. We assign the address of the constraints member function to the function pointer `x`, which causes the instantiation of the constraints function and checking of the concept's valid expressions. We then assign `x` to `x` to avoid unused variable compiler warnings, and wrap everything in a do-while loop to prevent name collisions.

```
#define REQUIRE(type_var, concept)                \
do {                                              \
  void (concept##_concept <type_var>::*x)() =     \
    concept##_concept <type_var>::constraints;    \
  x = x;                                          \
} while (0)
```

To check the type parameters of class templates, we provide the `CLASS_REQUIRES` macro which can be used inside the body of a class definition (whereas the `REQUIRES` macro can only be used inside of a function body). This macro declares a nested class template, where the template parameter is a function pointer. We then use the nested class type in a typedef with the function pointer type of the constraint function as the template argument. We use the `type_var` and `concept` names in the nested class and typedef names to help prevent name collisions.

```
#define CLASS_REQUIRES(type_var, concept)          \
  typedef void (concept##_concept <type_var>       \
    ::* func##type_var##concept)();                \
                                                   \
  template <func##type_var##concept FuncPtr>       \
  struct dummy_struct_##type_var##concept { };     \
                                                   \
  typedef dummy_struct_##type_var##concept<        \
    concept##_concept <type_var>::constraints>     \
    dummy_typedef_##type_var##concept
```

In addition, there are versions of `REQUIRE` and `CLASS_REQUIRES` that take more arguments, to handle concepts that include interactions between two or more types. `CLASS_REQUIRES` was not used in the implementation of the STL concept checks because several compilers do not implement template parameters of function pointer type.

The concept RandomAccessIterator defines the set of requirements that we would like to use to check the parameters of `stable_sort()`. Below we show how the complete concept checking class for this concept is constructed. The expressions within the

`constraints()` function correspond to the valid expressions section of the documentation for RandomAccessIterator. Typedefs can also be added to enforce the associated types of the concept. The RandomAccessIterator concept builds upon, or *refines* BidirectionalIterator, so we use the `REQUIRE` macro to invoke the concept checking class for that concept. By organizing the concepts in this way, we can reuse the concept checking classes such as `BidirectionalIterator_concept` and `LessThanComparable_concept` within the concept checking classes for more refined concepts such as RandomAccessIterator.

```
  template <class Iter>
  struct RandomAccessIterator_concept
  {
    void constraints() {
      REQUIRE(Iter, BidirectionalIterator);
      REQUIRE(Iter, LessThanComparable);
      REQUIRE2(typename std::iterator_traits<Iter>::iterator_category,
        std::random_access_iterator_tag, Convertible);
      typedef typename std::iterator_traits<Iter>::reference R;

      i += n;
      i = i + n; i = n + i;
      i -= n;
      i = i - n;
      n = i - j;
      i[n];
    }
    Iter a, b;
    Iter i, j;
    typename std::iterator_traits<Iter>::difference_type n;
  };
}
```

One potential pitfall in designing concept checking classes is using more expressions in the constraint function than necessary. For example, it is easy to accidentally use the default constructor to create the objects that will be needed in the expressions (and not all concepts require a default constructor). This is the reason we write the constraint function as a member function of a class. The objects involved in the expressions are declared as data members of the class. Since objects of the constraints class template are never instantiated, the default constructor for the concept checking class is never instantiated. Hence the data member's default constructors are never instantiated (C++ Standard Section 14.7.1 9).

## 4 Programming with Concepts

The process of deciding how to group requirements into concepts and deciding which concepts to use in each algorithm is perhaps the most difficult (yet most important) part of building a generic library. A guiding principle to use during this process is one we call the *requirement minimization principle*.

**Requirement Minimization Principle:** Minimize the requirements on the input parameters of a component to increase its reusability.

There is natural tension in this statement. By definition, the input parameters must be used by the component in order for the component to accomplish its task (by "component" we mean a function or class template). The challenge then is to implement the component in such a way that makes the fewest assumptions (the minimum requirements) about the inputs while still accomplishing the task.

The traditional notions of *abstraction* tie in directly to the idea of minimal requirements. The more abstract the input, the fewer the requirements. Thus, concepts are simply the embodiment of generic abstract data types in C++ template programming.

When designing the concepts for some problem domain it is important to keep in mind their purpose, namely to express the requirements for the input to the components. With respect to the requirement minimization principle, this means we want to minimize concepts.

It is important to note, however, that minimizing concepts does not mean simply reducing the number of valid expressions in the concept. For example, the `std::stable_sort()` function requires that the value type of the iterator be LessThanComparable, which not only includes `operator<()`, but also `operator>()`, `operator<=()`, and `operator>=()`. It turns out that `std::stable_sort()` only uses `operator<()`. The question then arises: should `std::stable_sort()` be specified in terms of the concept LessThanComparable or in terms of a concept that only requires `operator<()`?

We remark first that the use of LessThanComparable does not really violate the requirement minimization principle because all of the other operators can be trivially implemented in terms of `operator<()`. By "trivial" we mean one line of code and a constant run-time cost. More fundamentally, however, the use of LessThanComparable does not violate the requirement minimization principle because all of the comparison operators (<, >, <=, >=) are conceptually equivalent (in a mathematical sense). Adding conceptually equivalent valid expressions is not a violation of the requirement minimization principle because no new semantics are being added — only new syntax. The added syntax increases re-usability.

For example, the maintainer of the `std::stable_sort()` may some day change the implementation in places to use `operator>()` instead of `operator<()`, since, after all, they are equivalent. Since the requirements are part of the public interface, such a change could potentially break client code. If instead LessThanComparable is given as the requirement for `std::stable_sort()`, then the maintainer is given a reasonable amount of flexibility within which to work.

Minimality in concepts is a property associated with the underlying semantics of the problem domain being represented. In the problem domain of basic containers, requiring traversal in a single direction is a smaller requirement than requiring traversal in both directions (hence the distinction between ForwardIterator and BidirectionalIterator). The semantic difference can be easily seen in the difference between the set of concrete data structures that have forward iterators versus the set that has bidirectional iterators. For example, singly-linked lists would fall in the set of data structures having forward iterators, but not bidirectional iterators. In addition, the set of algorithms that one can implement using only forward iterators is quite different than the set that can be implemented with bidirectional iterators. Because of this, it is important to factor

families of requirements into rather fine-grained concepts. For example, the requirements for iterators are factored into the six STL iterator concepts (trivial, output, input, forward, bidirectional, and random access).

## 5 Concept Covering

We have discussed how it is important to select the minimal requirements (concepts) for the inputs to a component, but it is equally important to verify that the chosen concepts *cover* the algorithm. That is, any possible user error should be caught by the concept checks and not let slip through. Concept coverage can be verified through the use of *archetype classes*. An archetype class is an exact implementation of the interface associated with a particular concept. The run-time behavior of the archetype class is not important, the functions can be left empty. A simple test program can then be compiled with the archetype classes as the inputs to the component. If the program compiles then one can be sure that the concepts cover the component.

The following code shows the archetype class for the TrivialIterator concept. Some care must be taken to ensure that the archetype is an exact match to the concept. For example, the concept states that the return type of `operator*()` must be convertible to the value type. It does not state the more stringent requirement that the return type be `T&` or `const T&`. That means it would be a mistake to use `T&` or `const T&` for the return type of the archetype class. The correct approach is to create an artificial return type that is convertible to `T`, as we have done here with `input_proxy`. The validity of the archetype class test is completely dependent on it being an exact match with the concept, which must be verified by careful (manual) inspection.

```
template <class T>
struct input_proxy {
  operator T() { return t; }
  static T t;
};
template <class T>
class trivial_iterator_archetype
{
  typedef trivial_iterator_archetype self;
public:
  trivial_iterator_archetype() { }
  trivial_iterator_archetype(const self&) { }
  self& operator=(const self&) { return *this;  }
  friend bool operator==(const self&, const self&) { return true; }
  friend bool operator!=(const self&, const self&) { return true; }
  input_proxy<T> operator*() { return input_proxy<T>(); }
};

namespace std {
  template <class T>
  struct iterator_traits< trivial_iterator_archetype<T> >
  {
    typedef T value_type;
  };
}
```

Generic algorithms are often tested by being instantiated with a number of common input types. For example, one might apply `std::stable_sort()` with basic pointer types as the iterators. Though appropriate for testing the run-time behavior of the algorithm, this is not helpful for ensuring concept coverage because C++ types never match particular concepts, they often provide much more than the minimal functionality required by any one concept. That is, even though the function template compiles with a given type, the concept requirements may still fall short of covering the functions actual requirements. This is why it is important to compile with archetype classes in addition to testing with common input types.

## 6 Alternative Approaches

Concept checking tests whether certain expressions are valid and compile successfully. An alternative approach is to declare the signature that must be provided by the input. Such an approach is is used in Generic Java [3], Theta [5], and the Signatures extension to C++ [2].

Within the signature-based methods there are two approaches: the subtype approach and the where-clause approach. The subtype approach uses the equivalent of an abstract base class (or Java interface) to group the required method signatures. Base classes are used to place requirements on the input types in the component interface declaration. In C++ and Java, non-abstract classes must explicitly declare which abstract classes they implement, while in the Signatures C++ extension this is not necessary: the relationship is implicit, determined by the compiler at the function call site. The where-clause approach lists the required function signatures directly in the component interface, without grouping them into base classes [5].

The following is an example similar to one in D&E[8] that shows how the subtype-approach might look in C++. Typically a vector class would not require its element type to be comparable, but we include this to more closely follow Stroustrup's D&E example.

```
template <class T>
class Comparable<T> {
  bool operator==(const T&);
  bool operator<(const T&);
}

template <class T : Comparable<T> >
class vector { ... };
```

There are several problems with signature-based methods.

1. Signatures are not appropriate for C++ template programming because there is not a one-to-one relationship between expressions and the signatures that can implement the expression. For example, `operator++()` can be implemented as a free function or as a member function which have different signatures. It would be overly restrictive to list only one of the signatures as the interface requirement.

2. In some languages the subtype approach only applies to class types and not built-in types. The built-in type `int` is certainly Comparable, but it is not a class with member functions `operator==()` and `operator<()`.

3. Template programming is heavily based on parametric polymorphism, which is very different from subtype polymorphism. For example, one characteristic of type constraints for function templates is that they often involve functions with more than one argument, such as comparison operators. These kind of constraints are particularly difficult to express using subtyping [4], whereas they are quite natural to express with concept checks (which use parametric polymorphism).

The advantage of signatures is that they allow for the separate type-checking of the generic components , that is type-checking before instantiation of the template in a particular context. This negates the need for the more clumsy archetype classes described above. The signature-approach can also allow for separate compilation if dynamic dispatch is used.

## 7 Looking to the Future

Designing, implementing, and verify concept checks for generic C++ libraries must presently be done manually. As a result, the process is time-consuming and (likely to be) error-prone. Implementors would benefit greatly if some or all of this process could be automated.

A first step would be to have a tool that statically analyzes a class or function template and records all the kinds of expressions that involve the template parameter types. Such a tool would ease the task of verifying concept coverage. A second step would pattern match the set of all required expressions against a standard set (or library-defined set) of concepts, thereby summarizing the requirements in terms of concepts. This information could then be used in two ways. First it could be used to create readable reports for library documentation. Second, it could be used to provide informative compiler error messages without the need to manually insert concept checks.

Finally, we remark that there is much exciting work to be done in the general area of generic programming. Taking a step back from the task of providing better support for generic programming within the framework of the existing C++ language, there are a number of approaches one could take for designing a language that directly supports concepts.

## Acknowledgments

Thanks go to Alexander Stepanov for originating the idea to use function pointers to trigger the instantiation of concept checking code. Thanks also to Matthew Austern for establishing the concepts of the STL. We also acknowledge the many helpful suggestions from the reviewers. Parts of this work were performed while the first author was interning at SGI and while the second author was on sabbatical at Lawrence Berkeley National Lab. This work was partially funded by NSF grant ACI-9982205.

## References


[1] M. H. Austern. *Generic Programming and the STL*. Professional computing series. Addison-Wesley, 1999.

[2] G. Baumgartner and V. F. Russo. Signatures: A language extension for improving type abstraction and subtype polymorphism in C++. *Software–Practice and Experience*, 25(8):863–889, August 1995.

[3] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Addng genericity to the java programming language. In *OOPSLA*, 1998.

[4] K. B. Bruce, L. Cardelli, G. Castagna, the Hopkins Objects Group, G. T. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1:221–242, 1995.

[5] M. Day, R. Gruber, B. Liskov, and A. C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *OOPSLA*, pages 156–158, 1995.

[6] D. R. Musser and A. Saini. *STL tutorial and Reference Guide*. Addison-Wesley, Reading, 1996.

[7] A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project, May 1994.

[8] B. Stroustrup. *Design and Evolution of C++*. Addison-Wesley, 1994.


## 8 Appendix

### 8.1 Error messages using KAI C++ (with EDG frontend)

The following is an excerpt from the error message when compiling `bad_error_eg.cpp` with KAI C++ and without concept checking.

```
"algorithm", line 1274: error:
  no instance of overloaded function "std::min" matches the argument list
  argument types are: (std::iterator_traits<std::list<foo,
                       std::allocator<foo>>::iterator>::difference_type,
                       <error-type>)
      Distance len = min(p.second, Distance(last-first));
                     ^
  detected during instantiation of "void
     std::stable_sort(RandomAccessIterator, RandomAccessIterator)"
```

The following is an excerpt from the error message when compiling `bad_error_eg.cpp` with KAI C++ and with concept checking.

```
"concept_checks.h", line 248: error:
 no operator "<" matches these operands
 operand types are: std::list<foo, std::allocator<foo>>::iterator <
  std::list<foo, std::allocator<foo>>::iterator
```

```
        r = a < b || a > b || a <= b || a >= b;
                ^
detected during:
  instantiation of "void LessThanComparable_concept<Iter>::constraints()"
    at line 373
  instantiation of "void RandomAccessIterator_concept<Iter>::constraints()"
    at line 9 of "bad_error_eg.cpp"
  instantiation of "void std::stable_sort(RandomAccessIterator,
                      RandomAccessIterator)"
```

### 8.2 Error messages using Microsoft Visual C++

The following is an excerpt from the error message when compiling `bad_error_eg.cpp` with Microsoft Visual C++ and without concept checking.

```
algo(1164) : error C2784: '_Distance __cdecl std::operator -(
  const class std::reverse_iterator<_RandomAccessIterator,_Tp,
    _Reference,_Pointer,_Distance> &,
  const class std::reverse_iterator<_RandomAccessIterator,_Tp,
    _Reference,_Pointer,_Distance> &)'
  : could not deduce template argument for
  'const class std::reverse_iterator<_RandomAccessIterator,_Tp,
        _Reference,_Pointer,_Distance> &'
   from 'struct std::_List_iterator<struct foo,struct
     std::_Nonconst_traits<struct foo> >'
algo(1452) : see reference to function template instantiation
     'void __cdecl std::__inplace_stable_sort(
       struct std::_List_iterator<struct foo,
         struct std::_Nonconst_traits<struct foo> >,
       struct std::_List_iterator<struct foo,
         struct std::_Nonconst_traits<struct foo> >)'
   being compiled
```

The following is an excerpt from the error message when compiling `bad_error_eg.cpp` with Microsoft Visual C++ and with concept checking.

```
concept_checks.h(341) : error C2676: binary '+=' :
  'struct std::::_List_iterator<struct foo,
      struct std::::_Nonconst_traits<struct foo> >'
  does not define this operator or a conversion to a type acceptable
  to the predefined operator
concept_checks.h(332) : while compiling class-template member function
  'void __thiscall RandomAccessIterator_concept<
    struct std::::_List_iterator<struct foo,
          struct std::::_Nonconst_traits<struct foo> >
    >::constraints(void)'
```