Document number: N3351=12-0041
Date: 2012-01-13
Working group: Evolution
Reply to: Bjarne Stroustrup <bs@cs.tamu.edu>
Andrew Sutton <asutton@cs.tamu.edu>

A Concept Design for the STL

B. Stroustrup and A. Sutton (Editors)

Jan, 2012

Participants:
Ryan Ernst, A9.com, Inc.
Anil Gangolli, A9.com, Inc.
Jon Kalb, A9.com, Inc.
Andrew Lumsdaine, Indiana University (Aug. 1-4)
Paul McJones, independent
Sean Parent, Adobe Systems Incorporated (Aug. 1-3)
Dan Rose, A9.com, Inc.
Alex Stepanov, A9.com, Inc.
Bjarne Stroustrup, Texas A&M University (Aug. 1-3)
Andrew Sutton, Texas A&M University
Larisse Voufo [†], Indiana University
Jeremiah Willcock, Indiana University
Marcin Zalewski [†], Indiana University

Abstract

This report presents a concept design for the algorithms part of the STL and outlines the design of the supporting language mechanism. Both are radical simplifications of what was proposed in the C++0x draft. In particular, this design consists of only 41 concepts (including supporting concepts), does not require concept maps, and (perhaps most importantly) does not resemble template metaprogramming.

Contents

[†]Participated in editing of this report.

1 Introduction

The report is based on the result of a meeting August 4-9, 2011 hosted by A9.com in Palo Alto. Initiative for the meeting came from Andrew Lumsdaine of Indiana University, and the goal of the meeting was to produce a good conceptual definition of the STL and the language used to describe it.

The design we present is based on an initial draft specification of the algorithm sections of the STL and the concepts needed by that specification (written by Alex Stepanov, Dan Rose, and Anil Gangolli). The draft is clearly based on the influential book, *Elements of Programming* by Stepanov and McJones (2009), which describes the application of "the deductive method to programming by affiliating programs with the abstract mathematical theories that enable them to work." The design applies these ideas to the STL using the C++11 programming language, and addresses issues not dealt with in EoP: the rich C++ type system, move semantics, lambda expressions, and heterogeneous algorithm argument types. This design is (eventually) intended for ISO standardization.

In addition to the initial draft of the concepts and algorithms contained in this report that was prepared by our A9.com hosts, Alex Stepanov mailed out two papers to the participants ahead of the meeting:

1. "An Implementation of C++ Concepts in Clang" by Voufo et al. (2011). This paper presents a branch of the Clang compiler that is designed to support language feature prototyping, especially for concept-related features. The initial version of the compiler focused on features related to the C++0x proposal.

2. "Design of Concept Libraries for C++" by Sutton and Stroustrup (2011). The paper discusses concept design for C++ libraries and lays out a set of fundamental concepts that could be used as a basis for the STL. The design emphasizes the difference between purely syntactic requirements, called constraints, and concepts, which include both syntax and semantics.

These papers address some issues that were discussed during the meeting.

1.1 Motivation

The subject of this report are concepts for STL. Although concepts are familiar to much of our audience, we give a brief background to make this report complete and readable for a person that has not heard about concepts before.

A concept is a predicate that expresses a set of requirements on types. These requirements consist of syntactic requirements, which what related types, literals, operations, and expressions are available, and semantic requirements that give meaning to the required syntax and also provide complexity guarantees. Concepts are the basis of generic programming in C++ and allow us to write and reason about generic algorithms and data structures by constraining template arguments.

Concepts are not new to C++; the idea of stating and enforcing type requirements on template arguments has a long history (several methods are discussed in *The Design and Evolution of C++* (Stroustrup, 1994, ch. 15.4). Concepts were a part of documentation of the STL and are used to express requirements in the C++ standard (C++ Standards Committee, 2011). For example, Table 1 shows the definition of the STL InputIterator concept; it describes the requirements on types that would be used to iterate over and access (read) a sequence of elements. The first column lists expressions that must be valid (usable) for every input iterator, and the second gives their result types. The third and the fourth column describe the semantics of the expressions. Additionally, any type that would be an InputIterator must also satisfy the requirements of the Iterator and EqualityComparable concepts (not pictured).

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`a != b`	contextually convertible to `bool`	`!(a == b)`	pre: (a, b) is in the domain of ==.
`*a`	convertible to `T`		pre: `a` is dereferenceable. The expression `(void)*a, *a` is equivalent to `*a`. If `a == b` and `(a,b)` is in the domain of `==` then `*a` is equivalent to `*b`.
`a->m`		`(*a).m`	pre: `a` is dereferenceable.
`++r`	`X&`		pre: `r` is dereferenceable. post: `r` is dereferenceable or `r` is past-the-end. post: any copies of the previous value of `r` are no longer required either to be dereferenceable or to be in the domain of `==`.
`(void)r++`			equivalent to `(void)++r`
`*r++`	convertible to `T`	`{ T tmp = *r; ++r; return tmp; }`	

Table 1: The InputIterator concept from the C++ standard

Currently, it is conventional to name template arguments with their corresponding concepts. For example, the standard specifies the find algorithm as:

```
template<typename InputIterator, typename T>
InputIterator find(InputIterator first, InputIterator last, const T& value);
```

Here, the InputIterator template argument should satisfy the requirements listed in Table 1. Instantiating the algorithm over a type that does meet the stated requirements should fail to compile. Unfortunately, instantiation failures can only be caught at the point of failure. For example, if find is called as find(1, 5, 2), the compiler will emit the message, "int does not provide a unary operator*" (or something similar) and refer to the location in the program that triggered the lookup error. This can lead to very long and indecipherable error messages. One of the main reasons for including concepts as a part of the C++ language is to improve type checking for templates and generate better error messages.

An initial draft of concepts was designed for C++0x that did address many of the problems above. For example, the C++0x declaration of find is:

```
template<InputIterator Iter, typename T>
requires EqualtiyComparable<Iter::value_type, T>
Iter find(Iter first, Iter last, const T& value);
```

In this design, calling find(1, 5, 2) would result in an error message such as, "no concept map found for InputIterator<int>," indicating that int fails some of the requirements of the InputIterator concept. These instantiation errors can be caught at the point of use rather than failure, making error messages more readable.

The design has its flaws. For example, the C++0x specification of unique_copy is:

```
template<InputIterator InIter, class OutIter,
        EquivalenceRelation<auto, InIter::value_type> Pred>
requires OutputIterator<OutIter, RvalueOf<InIter::value_type>::type>
    && HasAssign<InIter::value_type, InIter::reference>
```

```
    && Constructible<InIter::value_type, InIter::reference>
    && CopyConstructible<Pred>
OutIter unique_copy(InIter first, InIter last, OutIter result, Pred pred);
```

The requirements of the algorithm describe what syntax is used in the implementation more than the abstractions required of the template parameters. In that sense, the design breaks encapsulation. This is one of the major problems that we seek to fix in this work. We want requirements to be terse and readable.

Furthermore, the C++0x made little use of axioms. Only four concepts (EqualityComparable, LessThanComparable, EquivalenceRelation, and StrictWeakOrdering) had associated semantics. Little if any meaning was attached to the syntax required by the concepts in the design. We feel that this is antithetical to the ideas of generic programming. It isn't possible to reason about the behavior of a program when you don't know what its symbols mean.

We have given only the briefest description of concepts and their role in C++. In order to help understand the issues and choices made in this design, readers should have an understanding of the designs provided in the C++0x draft:

1. Working Draft, Standard for Programming Language C++. WG21 N2914=09-0104 (C++ Standards Committee, 2009).

We also assume some acquaintance with the literature related to concepts, notably (and in chronological order):

1. "Specifying C++ Concepts" by Dos Reis and Stroustrup (2006).
2. "Concepts: Linguistic Support for Generic Programming in C++" by Gregor et al. (2006).
3. "Axioms: Semantics Aspects of C++ Concepts" by Dos Reis et al. (2009).
4. "The C++0x "Remove Concepts" Decision" by Stroustrup (2009).

The concept design and the corresponding language are written using the C++11 programming language, which is defined in ISO/IEC international C++ standard (C++ Standards Committee, 2011). We use features such as decltype, auto and type aliases freely. We do not think that this design could be elegantly expressed in C++98.

1.2 Approach

The "meeting rules" were stated at the outset by Alex Stepanov:

Objective: Provide a good conceptual definition for the STL by the end of the week.

Approach: Specify only the concepts needed for the STL. A concept we will define should either be used directly in an STL algorithm or be required to specify such a concept. To limit our task we start only at the algorithms in Clause 25 (Algorithms library). This entailed:

1. Reviewing algorithm signatures going from more complex (sort) to simple (find).
2. Reviewing the list of concepts in the initial design, their names, and their semantics.
3. Reviewing the syntax of bodies of concept requirements.
4. Reviewing the syntax of algorithm declarations (preferably including the pre- and post-conditions)

In a sense, we are designing concepts from scratch. We chose not to build on previous work because we feel that the approach taken was backwards. In C++0x, concept design focused on describing and designing language features, but not how they could be used to describe

requirements in elegant and simple ways. We have approached the problem from the opposite direction: write requirements first and designing the language to suit only our immediate needs.

The language for describing concepts will include only features needed to achieve the previously stated goals. This means:

- The design does not need concept maps so they do not appear as a language feature in this report.
- The design does not require the disjunction of requirements.
- The design does not distinguish between concepts and constraints as in the paper by Sutton and Stroustrup (2011).

These features and others can be added later, if needed.

1.3 Design Ideals

People arrived with a variety of ideals for the concepts and the language mechanisms. These include:

1. The concepts for the STL must be mathematically and logically sound. By this, we mean to emphasize the fact that we should be able to reason about properties of programs (e.g., correctness) with respect to the semantics of the language and the types used in those programs.
2. The concepts used should express general ideas in the application domain (hence the name "concepts") rather than mere programming language artifacts. Thinking about concepts as a yet another "contract" language can lead to partially formed ideas. Contracts force programmers to think about requirements on individual functions or interfaces, whereas concepts should represent fully formed abstractions.
3. The concepts should specify both syntactic and semantic requirements ("concepts are all about semantics"—Alex Stepanov). A concept without semantics only partially specifies an interface and cannot be reasoned about; the absence of semantics is the opposite of soundness ("it is insanity"—Alex Stepanov).
4. Symbols and identifiers should be associated with their conventional meanings. Overloads should have well defined semantics and not change the usual meaning of the symbol or name.
5. The concepts as used to specify algorithms should be terse and readable. An algorithm's requirements must not restate the syntax of its implementation.
6. The number of concepts used should be low, in order to make them easier to understand and remember.
7. An algorithm's requirements must not inhibit the use of very common code patterns in its implementation.
8. An algorithm should not contain requirements for syntax that it does not use, thereby unnecessarily limiting its generality.
9. The STL with concepts should be compatible with C++11 except where that compatibility would imply a serious violation of one of the first two aims.

Figure 1: Template requirements can be expressed in a range of ways, from exact syntactic patterns to regular, abstract concepts.

These ideals were articulated by various people in the meeting, but not specifically enumerated, and the design produced during the meeting attempts to satisfy these ideals.

Unfortunately, there are several conflicting goals that make it virtually impossible to meet all of these expectations. In particular, the idea that requirements should allow common code patterns but also be exact are at opposite ends of the spectrum shown in Fig. 1.

Every generic library design must choose the style in which it describes template requirements. The ways in which requirements are specified has a direct impact on the design of the concepts used to express them, and (as always) there are direct consequences of that choice.

For example, we could choose to state template requirements in terms of the exact syntax requirements of the template. This leads to concept designs that have large numbers of small syntactic predicates (e.g. HasPlus, HasComma, etc.). The benefit of this style of constraint is that templates are more broadly adaptable: there are potentially more conforming types with which the template will interoperate. On the downside, exact requirements tend to be more verbose, decreasing the likelihood that the intended abstraction will be adequately communicated to the library's users. The C++0x design is, in many aspects, a product of this style.

On the other end of the spectrum, we could choose to express requirements in terms of the required abstraction instead of the required syntax. This approach can lead to (far) fewer concepts in the library design because related syntactic requirements are grouped to create coherent, meaningful abstractions. Requirements can also be expressed more tersely, needing fewer concepts to express a set of requirements that describe how types are used in an algorithm. The use of abstract concepts also allows an algorithm to have more conforming implementations, giving a library author an opportunity to modify (i.e. maintain) a template's implementation without impacting its requirements. The obvious downside to this style is that it over-constrains templates; there may be types that conform to a minimal set of operations used by a template, but not the full set of operations required by the concept. The concepts presented in *Elements of Programming* approach this end of the spectrum.

The design presented in this report is somewhere in between. Clearly, we aim to emphasize abstraction and clarity over exact requirements. However, we see this as a starting point from which we might choose to strengthen template requirements. We know, for example, that there are opportunities to better support interoperability with non-copyable types, and we present this as an alternative design in Appendix D. However, that design also adheres to the goals and ideals stated above; it just prioritizes them differently.

1.4 Organization

This report is organized roughly like the meeting that produced it:

1. The declarations of the algorithms of the STL, with requirements on template arguments specified in terms of concepts. The concepts are informally described at their first use.

2. The definition of the concepts used to specify the algorithms. The language features used to define the concepts are informally described at their first use.

3. A set of appendices describing, in greater detail, the supporting language design and mechanics, preconditions and postconditions and alternative designs addressing issues and defects discovered during this work.

This is a top-down approach emphasizing what we primarily want: good specifications of algorithms. The concepts are defined to serve that end. The language features are defined to provide good specifications of concepts, as needed by the STL. If you prefer a bottom-up approach (e.g., language features before their uses), you can try to read the major sections of this report in reverse order (starting with Appendix A), but we don't recommend that.

The appendices of this report include:

1. A summary of the language features and mechanics defining and using concepts (Appendix A).

2. An index cross-referencing concepts and algorithms and concept diagrams showing dependencies between concepts (Appendix C).

3. Preconditions and postconditions for the algorithms described in this report (Appendix B).

4. An alternative concept design emphasizing more exact requirements on STL algorithms (Appendix D).

2 Algorithms

We start with algorithms because it is algorithms we want to specify cleanly, precisely, completely, and readably. If we can specify algorithms well, our concepts and the language mechanisms we use to specify the concepts are adequate. If not, no amount of sophistication in language mechanisms will help us.

We begin by discussing some preliminary ideas about iterators and ranges that are common to all algorithms. The algorithms in the STL, by and large, operate on iterator ranges. A *range*, or more specifically a *bounded range* is written using interval notation: [first, last), for example. This denotes a sequence of iterators that can be traversed, starting with first, and reaching last through repeated increment operations (++). The last iterator in a bounded range is sometimes called its *limit* (as in "up to but not including"). However, the name last is used more conventionally in specifications of the STL; we follow suit.

More formally, a bounded range is described thus: for every iterator i in the range [first, last) except last, the operation ++ is valid. Incrementing last is not guaranteed to result in a valid iterator. Bounded ranges let us write loops like this:

```
while(first != last) {
  // do something
  ++first;
}
```

We can, for example, write an algorithm to determine the distance between two iterators, first and last.

```
template<typename Iter>
int distance(Iter first, Iter last)
{
  int n = 0;
  while(first != last) {
```

```
    ++n;
    ++first;
  }
  return n;
}
```

This is roughly the same as the distance algorithm in the STL, although it is unconstrained and returns int instead of an an associated distance type. For convenience, we often document distances between iterators using subtraction. That is, writing last - first is equivalent to distance(first, last).

Every algorithm in the STL that takes a pair of iterators first and last has an implied precondition that [first, last) defines a valid bounded range. This ensures that ++ can be validly applied to first at most last - first times. The precondition applies to algorithms with other argument names, too:

- first1 and last1
- first2 and last2

Many algorithms in the STL do not specifically provide a second iterator as the limit of the of the range, but they nonetheless require that the unpaired iterator can be incremented some number of times. A range constructed from a starting iterator, say first, and an integral distance, say n, is called a *weak range*. With weak ranges, we can write loops like this:

```
while(n > 0) {
  // do something
  ++first;
  --n;
}
```

This is essentially an implementation of the STL advance algorithm for input and forward iterators:

```
template<typename Iter>
void advance(Iter& first, int n)
{
  while(n > 0) {
    ++first;
    --n;
  }
}
```

The next function can be written in terms of advance.

```
template<typename Iter>
Iter next(Iter i, int n = 1)
{
  advance(i, n);
  return i;
}
```

We highlight these functions because they are important operations for iterators. They define repeated applications (i.e. orbits) of an iterator's increment operator. When documenting algorithms, we often use addition to denote the repeated application of the increment operator, even if the actual iterator type does not support the syntax. For example, writing first + n is equivalent to writing next(first, n). The EoP book uses a special notation for differentiating weak and bounded ranges. In this report, we adopt the notation used in the standard and write them as bounded ranges: [first, first + n).

EoP also differentiates between weak ranges and counted range. A *counted range* is a weak range that has no cycles. A bounded range [first, last) is a counted range where first can be incremented n times and last == first + n.

As with algorithms taking bounded ranges as arguments, every algorithm taking an unpaired iterator has an implied precondition of a counted range. Algorithms ending with _n take a range [first, first + n). Every algorithm with an unpaired iterator first2 (e.g. copy) has an implied requirement on the range [first2, first2 + (last1 - first1)). Similar rules apply for arguments with unpaired iterators with different names, in particular result arguments. This does not apply to algorithms taking an unpaired iterator representing the midpoint of a sequence such as the rotate algorithm.

Note that unlike the loop constructed with bounded ranges, loops on weak ranges do not compare iterators using !=. Although a seemingly minor point, the fact that we don't use a particular operator (or set of operators) in the algorithm impacts its requirements. The requirements for an algorithm are derived from the set of operations that it uses, no more and no less. We don't try to infer requirements about intended usage, either. An algorithm's requirements only reflect what is actually necessary for its implementation.

2.1 Non-modifying Sequence Operations

This family of algorithms evaluates properties of sequences of objects. In particular, these algorithms test properties using equality and predicate functions. Testing a property of an object requires reading from an iterator by dereferencing it. For example, for a unary predicate function p, and iterators i and j, we could write:

```
p(*i)     // testing using a predicate
*i == *j  // testing two referenced objects for equality
```

Every algorithm in this section reads iterators in a similar way. We call iterators that can be read *input iterators*. An algorithm taking a bounded range of input iterators is required to be readable everywhere except at its limit. That is, for all iterators i in the range [first, last), the expression *i is valid, except when i == last. This is a precondition of every algorithm taking a range (bounded or weak) of input iterators.

2.1.1 All, Any, and None

The algorithms all_of, any_of, and none_of evaluate whether all, any, or no elements in a range satisfy a given property, represented by a predicate function. The declaration of all_of is pretty obvious:

```
template<InputIterator I, Predicate<ValueType<I>> P>
bool all_of(I first, I last, P pred);
```

Instead of a plain typename, we introduce template arguments with the concepts that describe their requirements. Here, the first argument I is required to be an InputIterator and the second argument P is required to be a Predicate taking a single argument of type ValueType<I>. InputIterator and Predicate are concepts.

All we have done here is to provide some syntax for precisely and tersely saying what the standard requires for the all_of algorithm. The use of a concept as a "template argument type" is a shorthand for mentioning the concept for the argument type as a requirements clause. We could have equivalently declared all_of like this:

```
template<typename I, typename P>
requires InputIterator<I> && Predicate<P, ValueType<I>>
bool all_of(I first, I last, P pred);
```

The template argument types that were used in the declaration above have been rewritten as a *requires clause*: a conjunction of concept requirements. Each requirement is evaluated against the template's arguments and their associated types. When a concept is used as a template argument type, the declared template argument is used as the first argument of the concept when written as part of a requires clause.

- "InputIterator I" becomes InputIterator<I>.
- "Predicate<ValueType<I>> P" becomes Predicate<P, ValueType<I>>.

An InputIterator is a kind of iterator that supports forward traversal (++) and reading the value by dereferencing (unary *). The reading of dereferenced values is required by all Readable iterators, and every InputIterator is Readable.

P must be a Predicate function taking a ValueType<I>. By convention, all function objects in the STL are required to be copy- and move-constructible. Copy and move assignment are not required, nor is default construction.

ValueType<I> is a template alias that refers to the value type associated with a Readable iterator. It replaces the use of typename iterator_traits<I>::value_type everywhere it would normally appear in the STL. The use of template aliases as a replacement for type traits dramatically simplifies the specification of requirements and algorithm signatures.

Note that we might have declared the function as:

```
template<typename I, typename P>
requires Predicate<P, ValueType<I>> && InputIterator<I>
bool all_of(I first, I last, P pred);
```

However, this may yield unexpected compiler errors. The evaluation of requirements is exactly the same as the evaluation of Boolean conjunctions: left to right. In this last example, the alias ValueType<I> is referenced before the concept that requires it: InputIterator. If we try to instantiate the algorithm with a non-conforming type, say int, we may not get the graceful error message we want. Instead of an error informing us that, "int is not an InputIterator", we might get the less obvious error, "ValueType<int> does not name a valid type".

The order in which requirements are evaluated is important. Requirements listed as the type of template arguments are also evaluated from left to right and before the requires clause. A concept should be checked before any of its associated types or functions are referenced in the constrained template.

We are of the opinion that the first form (using concepts as template argument types) is by far most readable for users (those mythical "ordinary programmers"), if not necessarily for experienced type theorists. By convention, we use the most specific unary (single argument) concept describing the template argument as its "type". In the first declaration of all_of, the template argument I is most specifically described as an InputIterator, and P as a Predicate taking ValueType<I> as an argument. As matter of style, we prefer not to have "naked" type names introducing template parameters when we can state their actual requirements.

We use an algorithm, such as all_of, exactly as in C++11. The only difference is that errors are caught immediately at the call point. For example:

```
bool is_all_clear(vector<int>& v)
{
  return all_of(v.begin(), v.end(), is_zero);
}
```

The vector iterators are InputIterators, so the first two arguments to all_of are acceptable. They are, of course RandomAccessIterators, but as usual a RandomAccessIterator can be used where an InputIterator is required (§3.5). If is_zero is a function (or a function object) that takes an argument of a type to which int can be converted, and it returns something that can be converted

to bool, the instantiation of the call to all_of succeeds; if not, we get an immediate compile-time error because of the requirements on the algorithm.

Note that this definition of all_of does not require homogeneous types; the Predicate requirement accommodates conversion of argument types (§3.4). For example:

```
bool is_zero(long long x)
{
  return x==0;
}
```

This is an acceptable function argument for the all_of declared above.

The type requirements and preconditions of any_of and none_of are identical to those of all_of.

```
template<InputIterator I, Predicate<ValueType<I>> P>
bool any_of(I first, I last, P pred);

template<InputIterator I, Predicate<ValueType<I>> P>
bool none_of(I first, I last, P pred);
```

Note that *Elements of Programming* calls the any_of algorithm, some Stepanov and McJones (2009). This more closely aligns with conventional vocabulary of quantified expressions; "for some x... " is usually preferred over "for any x..." We think some_of would be a better name for the algorithm. This is reflected in the quantifier notation presented in Appendix B.

The Predicate concept doesn't just require that we can call pred with the value read from an iterator *i; it requires that the function is *equality preserving*. That is, calling pred with equal values will always yield equal results. For any objects x and y, if x==y, then pred(x)==pred(y), regardless of the state of the program.

This doesn't mean that pred has to be a functionally pure operation; side effects are allowed as long as they don't affect the computation in such a way that pred(x)==pred(x) can be false. For example, the function object non_null satisfies the requirements of the Predicate concept in the following program.

```
int tests = 0;

template<typename T>
struct non_null
{
  bool operator(T const* p) const
  {
    ++tests;
    return p != nullptr;
  }
};
```

However, a function that returns true or false based on a randomized coin flip is not a Predicate. The equality preserving property supports our ability to reason equationally about programs. We can't expect concepts to be mathematically sound if we can't support equational reasoning. There are many useful cases, however, where equality preservation is not required.

2.1.2 For Each

The for_each algorithm applies a function to each element of a bounded range. Its declaration is:

```
template<InputIterator I, Function<F, ValueType<I>> F>
F for_each(I first, I last, F f);
```

As with the all_of, any_of and none_of algorithms, this algorithm requires I to be an InputIterator and a F to be a unary Function taking a ValueType<I> argument. Unlike the previous algorithms, this one does not require F to be a predicate because the result of f is not used in a Boolean context; in fact, the result is not used at all. F could have void as the result type, and it often is in practice.

This algorithm is an example where a function is not required to preserve equality. The Function concept, allows (and in fact expects) the expression f(*i) to have some side effects. The result of f's application is also ignored by the algorithm so there are no explicit dependencies on its results. A perfectly reasonable template argument substitution for F would be a function object that assigns a random value to each iterated object in the range (although generate might be a better choice of algorithm for that application).

2.1.3 The Find Family

The following algorithms search for values in a range of iterators. These algorithms rely on (and generalize) the equality comparison of values.

```
template<InputIterator I, EqualityComparable<ValueType<I>> T>
I find(I first, I last, const T& value);
// not fully compatible
```

The find algorithm searches for a value in the range [first, last) by comparing each iterator i in that range to value using the expression *i == value. Obviously, I is required to be an InputIterator, and T is required to be EqualityComparable with ValueType<I>.

The EqualityComparable concept is *overloaded* to describe requirements on a single type (e.g. EqualityComparable<T>) or two types (e.g. EqualityComparable<T, ValueType<I>> (§3.3). We use the latter version here since T and ValueType<I> are not required to be the same.

Note that this two-argument overload of EqualityComparable §3.3 does not simply mean that an overload of == is available for the two types; this is not the same as HasEqual in the C++0x proposal (C++ Standards Committee, 2009). The two-argument version of this concept generalizes the one-argument version for use with different types and formally defines the meaning of the required syntax. The EqualityComparable requires

- that T and ValueType<I> are both EqualityComparable,
- that T and ValueType<I> share a CommonType C §3.2,
- that C is EqualityComparable, and
- that (semantically) any equality comparison between values of those different types can be replaced by an equivalent equality comparison on values of the common type.

This last requirement preserves the mathematical axioms of equality for types sharing a common type. The requirements of cross-type equality comparison are far stricter than might be expected. However, we justify the stronger requirements by providing a definition of the expression's meaning.

What kinds of different types can be compared for equality? For starters, every pair of built-in arithmetic types share a common type. For example, char and int have the common type int. int and double have the common type double. This means that programs like that below will continue to be valid using this design.

```
vector<double> v = {1.0, 2.0, 3.0};
find(v.begin(), v.end(), 3);
```

Equality comparisons between an int and a double requires that the int value be promoted to double, and the comparison carried out on the double values. The cross-type EqualityComparable

concept states that the equality comparison of different types (sharing a common type) is equivalent to converting to that common type and then performing the comparison. Not only is the program valid, it can now be shown to be mathematically sound because we can reason about the semantics of these cross-type comparisons in terms of the corresponding comparison on the common type.

There are, of course, limitations. What happens when the comparison is lossy? What if we want to compare a long long and a float? There's no guarantee that the value of a 64-bit long long integer can be faithfully represented by a 32-bit float. The short answer is that the C++ language describes the behavior of these comparisons (C++ Standard, conv). Unfortunately, this doesn't match the mathematical ideal, but at least the issue is well known and understood. We discuss this further in §3.2.

The requirements given here are not fully compatible with those in the C++11 standard. The standard allows the algorithm to be instantiated for any type arguments where *i==value is a valid expression. We have strengthened the requirements of the algorithm in accordance with our design ideals, namely that of mathematical soundness.

The strengthening of this requirement has the potential to break existing code. This is especially the case where programmers have overloaded == as a shorthand for testing equivalence of identity instead of true equality. For example, an employee object might be compared with a name (a string). It could easily be the case that two employees share the same name, which means that this is not the same as testing equality.

In the STL, the symbol == means equality. The use of == to compare objects of unrelated type assigns unusual meaning to that symbol. Generic programming is rooted in the idea that we can associate semantics with operations on objects whose types are not yet specified. This gives us the ability to say, with confidence, that an operation has the desired effect even though we don't know the concrete types of the operands. Mathematically speaking, equivalence is defined for objects of a single type and must be reflexive, symmetric, and transitive. How can a predicate on two unrelated types be shown to satisfy the reflexive property, which is defined in terms of a single object? That is, how can we assign, to a single object, two different type arguments that would make an instantiation of the reflexive property true? How could we do the same for the transitive property, which is quantified over three objects?

Operations whose semantics cannot be mapped onto mathematical equations do not readily support equational reasoning. Our use of common types allows us to extend the semantics of equality comparison to related types. Cross-type relations and operations are used heavily in this design because they preserve some of the latitudes of the C type system, but base them in mathematical equations.

Although the semantics of the comparison are described in terms of conversion to the common type, no conversion is actually required within the algorithm. If the data type provides the correct overloads for the cross-type comparison, then those are used.

The correct way to implement comparisons on unrelated types is to use the right function; if you want to find an employee by name, use find_if and provide a function to compare the names. find_if generalizes the find algorithm and takes an arbitrary predicate as an argument.

```
template<InputIterator I, Predicate<ValueType<I>> P>
I find_if(I first, I last, P pred);

template<InputIterator I, Predicate<ValueType<I>> P>
I find_if_not(I first, I last, P pred);
```

Both algorithms are parameterized over an InputIterator and a unary Predicate function, P, that takes an argument of ValueType<I>. find_if returns the first iterator for which the predicate is true; find_if_not returns the first iterator for which it is not.

Finding employees by name can be achieved using find_if and an appropriate lambda expression, for example:

```
string name = "Smith";
list<employee> emps = { ... };
auto i = find_if(emps.begin(), emps.end(),
                 [&name](const employee& emp) { return emp.name() == name; });
```

The find_first_of algorithm searches the range [first1, last1) for a value that is, in some way, equivalent to one in the range [first2, last2). There are two overloads of this algorithm:

```
template<InputIterator I1, ForwardIterator I2>
requires EqualityComparable<ValueType<I1>, ValueType<I2>>
I1 find_first_of(I1 first1, I1 last1, I2 first2, I2 last2);

template<InputIterator I1, ForwardIterator I2, Predicate<ValueType<I1>, ValueType<I2>>  P>
I1 find_first_of(I1 first1, I1 last1, I2 first2, I2 last2, P pred);
```

The first overload searches for a value in [first1, last1) that is equal to one of the values in [first2, last2). The test for equality is written using the operator,==, and in particular, this algorithm compares an iterator i from the first range and an iterator j from the second using *i == *j. The second overload generalizes the use of == to any binary predicate. Instead of using ==, it compares pred(*i, *j). This pattern of overloading and generalization is repeated throughout the STL.

- An algorithm is initially defined in terms of the syntax of a particular operator. Here, the first overload is implemented in terms of ==.
- A generalized overload replaces the concrete syntax with function parameters. The second overload replaces == with pred, which could have any conceivable meaning.

In essence, this is the same pattern as find and find_if. We have to use two names in that case since the overloads couldn't otherwise be reliably distinguished.

Both find_first_of algorithms requires I2 to be a ForwardIterator. A ForwardIterator is an InputIterator that supports multiple passes over a range (§3.5.3). This is required by the algorithm since the range [first2, last2) is traversed for every iterator in [first1, last1).

In the first algorithm, cross-type equality of the value types of I1 and I2 is required by the EqualityComparable concept. Again, this specification is not fully compatible with the standard.

The second overload requires a binary Predicate taking arguments types ValueType<I1> and ValueType<I2> in that order. We do not require that pred can be called symmetrically. This is, for any objects a and b of different types, the fact that pred(a, b) is a valid expression does not imply that pred(b, a) is also valid.

Although the initial algorithm is specified in terms of equality, the generalized version does not require pred to be an equivalence relation (§3.3). The second overload could easily be used, for example, to compare objects of different type: find the first employee whose benefits include one of the following perks, {12 weeks vacation, company car, obscene pension}.

Note that concept names can be overloaded. Between the find_if and find_first_of algorithms, the Predicate concept is used in two different ways: find_if algorithm requires a unary Predicate, while find_first_of requires a binary Predicate. This style of overloading is based on the number of arguments passed to the concept.

The adjacent_find algorithm searches the range [first, last) for a pair of iterators. As with find_first_of, there are two overloads.

```
template<ForwardIterator I>
requires EqualityComparable<ValueType<I>>
I adjacent_find(I first, I last);

template<ForwardIterator I, Relation<ValueType<I>> R>
I adjacent_find(I first, I last, R comp);
```

The first overload finds the first consecutive pair of iterators with equal (==) values. The second overload generalizes equality to an arbitrary Relation on the value type. A Relation is a binary Predicate traditionally having homogeneous argument types. Our design also includes a cross-type Relation that accepts heterogeneous (but related) argument types (see §2.3.4 for an example of usage).

By convention we always choose to write binary Predicate requirements as Relation requirements when the argument types are the same. Writing, e.g. , Predicate<R, ValueType<I>, ValueType<I>> is unnecessarily verbose. Again, comp is not required to be an equivalence relation.

2.1.4 The Count Family

The count algorithms are logical extensions of the find algorithms. They count the number of iterators satisfying a condition, returning the number of instances.

```
template<InputIterator I, EqualityComparable<ValueType<I>> T>
DistanceType<I> count(I first, I last, const T& value);

template<InputIterator I, Predicate<ValueType<I>> P>
DistanceType<I> count_if(I first, I last, P pred);
```

Both functions have the result type DistanceType<I>, which is an alias for the iterator's associated distance type. The use of DistanceType<I> replaces the less terse typename iterator_traits<I>::difference_type. We use the name "distance" rather than "difference" because the type is more closely associated with the distance algorithm, rather than the subtraction of values.

Again, the first overload is not fully compatible with the standard. The use of cross-type equality limits viable instantiations to those where T and ValueType<I> share a common type in addition to being EqualityComparable.

2.1.5 Mismatch and Equal

The mismatch and equal algorithms compare the elements in two ranges, pairwise.

```
template<InputIterator I1, WeakInputIterator I2>
requires EqualityComparable<ValueType<I1>, ValueType<I2>>
pair<I1, I2> mismatch(I1 first1, I1 last1, I2 first2);

template<InputIterator I1, WeakInputIterator I2,  Predicate<ValueType<I1>, ValueType<I2>> P>
pair<I1, I2> mismatch(I1 first1, I1 last1, I2 first2, P pred);
```

The first2 function parameter is the first iterator in a weak range, [first2, first2 + (last1 - first1)). The algorithm does not compare iterators in this range using == or !=; they are only incremented. Because of this, the type requirement of I2 is WeakInputIterator. A WeakInputIterator is a generalization of InputIterators. They can be incremented and dereferenced, but they are not required to support equality comparison. The prefix "weak" in the concept names is specifically meant to associate them with weak ranges.

The equal algorithm has identical requirements and preconditions to mismatch.

```
template<InputIterator I1, WeakInputIterator I2>
requires EqualityComparable<ValueType<I1>, ValueType<I2>>
bool equal(I1 first1, I1 last1, I2 first2);

template<InputIterator I1, WeakInputIterator I2, Predicate<ValueType<I1>, ValueType<I2>> P>
bool equal(I1 first1, I1 last1, I2 first2, P pred);
```

2.1.6 Permutations

The is_permutation algorithm determines if one sequence is a permutation of another.

```
template<ForwardIterator I1, ForwardIterator I2>
requires EqualityComparable<ValueType<I1>, ValueType<I2>>
bool is_permutation(I1 first1, I1 last1, I2 first2);

template<ForwardIterator I1, ForwardIterator I2, Relation<ValueType<I1>, ValueType<I2>> R>
bool is_permutation(I1 first1, I1 last1, I2 first2, R comp);
// precondition: comp is an equivalence relation
```

The first overload is not fully compatible with the standard because of its requirement on cross-type equality. In this case, however, the requirement stated here is actually *weaker* than that stated in the standard. The use of common types to define a mathematically sound cross-type equality lets us generalize the algorithm for sequences of differently-typed values.

The second overload generalizes the equality comparison of the first overload as an equivalence relation (§3.3). Like EqualityComparable, the Relation concept is overloaded to accept arguments of different types. The semantics of the "cross-type" Relation are defined in terms of the common types of the arguments. This is also more general than required by the C++ standard (C++ Standard, alg.is_permutation).

As in the standard, this algorithm has quadratic complexity. A more efficient (i.e., $\mathcal{O}(n \log n)$) version of this algorithm relies on sorting. A trivial implementation is:

```
template<ForwardIterator I1, InputIterator I2>
requires Sortable<I1> && EqualityComparable<ValueType<I1>, ValueType<I2>>
bool is_permutation(I1 first1, I1 last1, I2 first2)
{
  vector<ValueType<I2>> tmp(first1, last1);
  sort(tmp.begin(), tmp.end());
  return equal(tmp.begin(), tmp.end(), first2);
}
```

The Sortable describes requirements on sortable iterator ranges (it effectively allows the use of the sort algorithm) and is defined in §3.6. In addition to the sorting requirements, this also requires copyability (through Sortable) but relaxes the ForwardIterator requirement on the I2. This implementation requires exactly n copies, at most n applications of ==, and is sorted in $\mathcal{O}(n \log n)$ time (where n is last - first).

The Relation overload would be:

```
template<ForwardIterator I1, InputIterator I2, Relation<ValueType<I1>, ValueType<I2>> R>
requires Sortable<I1, R>
bool is_permutation(I1 first1, I1 last1, I2 first2, R comp)
{
  vector<ValueType<I2>> tmp(first1, last1);
  sort(tmp.begin(), tmp.end(), comp);
  return equal(tmp.begin(), tmp.end(), first2,
               [](const ValueType<I1>& a, const ValueType<I2>& b) {
                 return !comp(a, b) && !comp(b, a);
               });
}
```

Here, equality is checked using the symmetric complement of the strict weak ordering R. The performance is on par with the previous overload except that it requires at most $2n$ applications of comp in order to compute equality.

Note that we could provide both the quadratic and quasilinear non-Relation versions of the algorithm within the same library. Concept-overloading will choose the more efficient version

for totally ordered, regular types, and fail back to the quadratic version for unordered types. Unfortunately, we cannot provide both versions of the Relation overloads. There is insufficient type information to distinguish an equivalence relation from a strict weak ordering.

Note that adding these overloads would also let us re-categorize the algorithm and place them with the permutation generators (§2.3.8) in the sorting algorithms. That seems to us to be a more natural categorization of the algorithm.

2.1.7 Search

The search algorithms include search, search_n, and find_end. We include find_end in the search family because it is more closely associated (by behavior and requirements) with these algorithms than with those in the find family.

```
template<ForwardIterator I1, ForwardIterator I2>
requires EqualityComparable<ValueType<I1>, ValueType<I2>>
I1 search(I1 first1, I1 last1, I2 first2, I2 last2);
// not fully compatible

template<ForwardIterator I1, ForwardIterator I2,
         Predicate<ValueType<I1>, ValueType<I2>> P>
I1 search(I1 first1, I1 last1, I2 first2, I2 last2, P pred);
```

The first overload of search is not fully compatible with the standard because of the requirement on cross-type equality. In the second overload, we require pred to be a binary Predicate accepting arguments of type ValueType<I1> and ValueType<I2> (in that order). The requirements for the find_end algorithm are identical to those of search:

```
template<ForwardIterator I1, ForwardIterator I2>
requires EqualityComparable<ValueType<I1>, ValueType<I2>>
I1 find_end(I1 first1, I1 last1, I2 first2, I2 last2);
// not fully compatible

template<ForwardIterator I1, ForwardIterator I2,
         Predicate<ValueType<I1>, ValueType<I2>> P>
I1 find_end(I1 first1, I1 last1, I2 first2, I2 last2, P pred);
```

The search_n algorithm is related to search except that it finds a subsequence of count equal values in the range [first, last).

```
template<ForwardIterator I, EqualityComparable<ValueType<I>> T>
I search_n(I first, I last, DistanceType<I> count, const T& value);
// different from standard

template<ForwardIterator I, typename T, Predicate<ValueType<I>, T> P>
I search_n(I first, I last, DistanceType<I> count, const T& value, P pred);
// different from standard
```

The search_n algorithm differs from the standard in two ways. First, the use of cross-type equality is a stronger requirement. Second, we have removed the usual Size template parameter and replaced it with DistanceType<I>. We do this to be more precise about the requirements of the algorithm. By replacing the template parameter Size with the associated DistanceType, we force callers of the function to promote argument types to an integer with the appropriate width.

What we really want to say is that the algorithm should not accept integer values whose types are wider than DistanceType<I>. As an extreme example, consider what's being asked for in this program:

```
vector<int> v = { /* ... */ };
auto i = search_n(v.begin(), v.end(), numeric_limits<long long>::max(), 0);
```

How could a vector, which can theoretically have at most 2^{32} objects, contain a subsequence of $2^{63} - 1$ consecutive 0's? Obviously, a correct implementation guards against such extreme cases, but we could do better by forcing a conversion at the call site using the declaration above. When compiling with conversion warnings enabled, we effectively warn users about mathematically unsound code. A recent version of Clang says:

```
warning: implicit conversion loses integer precision:
  'long long' to 'int' [-Wshorten-64-to-32]
```

In every algorithm taking a weak range that is explicitly bounded by a integral distance (i.e., the *_n algorithms), we replace the template Size parameter with the distance type of the range's iterator type. Realistically, the danger of allowing unsound instantiations of the search_n algorithm is minimal. However, algorithms like copy_n could potentially overflow their maximum representable distance if given large enough values for count.

2.2 Modifying Sequence Operations

In this section, we discuss requirements for algorithms that modify sequences of objects by writing to an iterator. Writing to an iterator typically means that we are copying a value into the object referenced by an iterator, but it could also mean that we are *moving* the value into the referenced object. Unless we explicitly describe a write as a move operation, we intend for it to mean a copy.

Every algorithm in this family takes an iterator range through which values are copied or moved. That is, for any iterator i in that range (except the limit), we can potentially write either or both of the following:

- *i = x; i is writable: copies the value of x to the object referenced by i.
- *i = move(x); i is move-writable; moves the value of x into the object referenced by i using the std::move function.

The kind of operation depends on the requirements of the algorithm and the type of x. In general, we refer to iterators that support this kind of use as *output iterators*.

An algorithm taking a bounded range of output iterators [first, last) has an implied precondition that *i = x is a valid operation for every i in [first, last) except when i == last. This precondition applies to algorithms taking weak ranges as well; [first, first + n) is writable everywhere except the iterator first + n. Similar preconditions are implied for algorithms that move values into a range of output iterators.

Note that unlike InputIterators, our design does not include a corresponding OutputIterator. We further discuss this in §3.7.

2.2.1 Copy

This family of algorithms copy the values in one (bounded) range [first, last) into another (weak) range [result, result + (last - first)).

```
template<InputIterator I, WeaklyIncrementable Out>
requires IndirectlyCopyable<I, Out>
Out copy(I first, I last, Out result);
```

The algorithm imposes two requirements on Out: WeaklyIncrementable and IndirectlyCopyable. We require WeaklyIncrementable because result is in a weak range, so a weak concept is sufficient.

As with input iterators, weakly incrementable types may "consume" any previous state when incremented. In fact, all InputIterators are actually WeaklyIncrementable (§3.5.2).

The IndirectlyCopyable concept requires that we can copy the value of an I iterator to an Out iterator. The concept is an alias for Writable<ValueType<I>, Out> (§3.5.1). The Writable concept is central to many algorithms in this section.

Note that we don't require WeakInputIterator<Out> because the algorithm doesn't actually *read* from any of the iterators in the output range. Dereferencing an iterator to read from it is entirely different than dereferencing an iterator in order to write to it. The concepts required by the algorithm reflect the use of types within the algorithm.

Unlike the STL and the C++ standard, this report does not include an explicit OutputIterator concept. Those requirements are subsumed by the Writable concept. Writability is a binary concept (a concept with two parameters) that expresses a relationship between an iterator type and a value type. It is not a property of the iterator itself, and the value type being written may be unrelated to the value type of the iterator if one exists at all. In copy, for example, Out is not required to be Readable and has no associated value type. In fact, writing ValueType<Out> in the declaration or body of copy should result in a compiler error. All we can say about Out is that we can indirectly copy the values in the range [first, last) into the output range.

The impact of this design on iterator implementations is positive. One concrete benefit of this design is that we do not need to associate a value type with iterators that do not actually have one. For example, ValueType<ostream_iterator<T>> is undefined in our design, not void as it is in the STL implementation and the standard specification.

```
template<WeakInputIterator I, WeaklyIncrementable Out>
requires IndirectlyCopyable<I, Out>
Out copy_n(I first, DistanceType<I> n, Out result);
// different from standard
```

The requirements for copy_n are similar to those of copy except that it takes a weak range [first, n). As with search_n in the previous section, we have replaced the free type parameter, Size, with DistanceType<I>. In this case, the change makes the algorithm more correct. By requiring conversion at the call site, we allow compilers to catch errors where the conversion to DistanceType<I> is lossy, assuming of course, that the proper warning flags are enabled during compilation.

The requirements of copy_if are a combination of the requirements for find_if and copy. Its declaration is:

```
template<InputIterator I, WeaklyIncrementable Out, Predicate<ValueType<I>> P>
requires IndirectlyCopyable<I, Out>
Out copy_if(I first, I last, Out result, P pred);
```

The copy_backward algorithm copies the elements of [first, last) into [result - (last - first), result) in reverse order. We declare it as:

```
template<BidirectionalIterator I, BidirectionalIterator Out>
requires IndirectlyCopyable<I, Out>
Out copy_backward(I first, I last, Out result);
```

The copy_backward algorithm requires both I and Out to be BidirectionalIterators. A BidirectionalIterator is a ForwardIterator that also supports decrement operations (--).

We generally assume that the result of a copy operation is that a) the original remains unmodified, and that b) the copy is equal to the original range of values. The input and output ranges may overlap so long as no values are read after they have been written.

Note that these algorithms can be instantiated in such a way that the input sequence is invalidated by the operation. Passing move_iterators as the input range will cause the algorithm to move (not copy!) the values into the output range. This will invalidate the original iterators; their referenced objects are left partially formed. Because we can do this, and because it is sometimes useful to do so, we do not require that copy preserves the original input.

2.2.2 Move

The move and move_backward algorithms are similar to copy and copy_backward, except in the requirement of Writability.

```
template<InputIterator I, WeaklyIncrementable Out>
requires IndirectlyMovable<I, Out>
Out move(I first, I last, Out result);

template<BidirectionalIterator I, BidirectionalIterator Out>
requires IndirectlyMovable<I, Out>
Out move_backward(I first, I last, Out result);
```

These algorithms require IndirectlyMovable, which is an alias for MoveWritable<ValueType<I>, Out>. The MoveWritable concept is analogous to the Writable concept, but replacing a copy assignment with a move assignment. That is, for any valid iterator i, the MoveWritable concept requires the syntax *i = move(x) to be well formed. Assuming that i is not the limit of a range, the expression moves the value of x into the object referenced by i, and leaves the object x partially formed.

2.2.3 Swap

The iter_swap and swap_ranges algorithms exchange the values in objects referenced by iterators and ranges of iterators, respectively.

```
template<Readable I1, Readable I2>
requires IndirectlyMovable<I2, I1> && IndirectlyMovable<I1, I2>
    && Semiregular<ValueType<I1>> && Semiregular<ValueType<I2>>
void iter_swap(I1 i, I2 j);
// different than standard

template<InputIterator I1, WeakInputIterator I2>
requires IndirectlyMovable<I2, I1> && IndirectlyMovable<I1, I2>
    && Semiregular<ValueType<I1>> && Semiregular<ValueType<I2>>
I2 swap_ranges(I1 first1, I1 last1, I2 first2);
// different than standard
```

The IndirectlyMovable concept (§3.5.1) describes the requirements of exchanging elements pointed to by different iterators. The Semiregular concept is required to create the temporary used to implement the swap as:

```
auto x = move(*i);
*i = move(*j);
*j = move(x);
```

Both value types are required to be Semiregular since an implementation could choose to construct a temporary from either *i (as above) or *j with equivalent behaviors. Note that if the value types of I1 and I2 are the same, then the these requirements also guarantee that the expression swap(*i, *j) is valid.

The use of Semiregular over-constrains these algorithms by requiring copies. We describe an alternative design that relaxes those requirements (bringing them into line with the standard) in Appendix D.

2.2.4 Transform

The transform algorithm applies a unary or binary function to the elements in a range. The term "operation" is used incorrectly in the standard. Operations generally refer to mathematical operations and have different syntactic and semantic requirements (§3.4).

```
template<InputIterator I, WeaklyIncrementable Out, Function<ValueType<I>> F>
requires Writable<ResultType<F, ValueType<I>>, Out>
Out transform(I first, I last,  Out result, F f);

template<InputIterator I1, InputIterator I2, WeaklyIncrementable Out,
         Function<ValueType<I1>, ValueType<I2>> F>
requires Writable<ResultType<F, ValueType<I1>, ValueType<I2>>, Out>
Out transform(I1 first1, I1 last1, I2 first2, Out result, F f);
```

As with for_each (§2.1.2), the transform algorithms accept Function arguments. The arity required by the function is given by the number of arguments passed to the concept; the first overload requires a unary Function, the second, a binary Function.

Both algorithms require that the result of the function be assigned through an iterator in the writable weak range [result, result + (last1 - first1)). This requirement is stated by the Writable requirement. In order to know the result type of F, we use the ResultType type function. This type function is defined for all Function types, and takes as arguments, the function type and its argument types.

The ResultType type function is semantically equivalent to the result_of type trait. For example, ResultType<F, ValueType<I>> means the same as typename result_of<F(ValueType<I>)>::type.

2.2.5 Replace

The replace algorithms replace one value in a sequence with another.

```
template<InputIterator I, EqualityComparable<ValueType<I>> T>
requires Writable<T, I>
void replace(I first, I last, const T& old_value, const T& new_value);

template<InputIterator I, Predicate<ValueType<I>> P, typename T>
requires Writable<T, I>
void replace_if(I first, I last, P pred, const T& new_value);

template<InputIterator I, WeaklyIncrementable Out, EqualityComparable<ValueType<I>> T>
requires IndirectlyCopyable<I, Out> && Writable<T, Out>
Out replace_copy(I first, I last, Out result, const T& old_value, const T& new_value);

template<InputIterator I, WeaklyIncrementable Out, Predicate<ValueType<I>> P, typename T>
requires IndirectlyCopyable<I, Out> && Writable<T, Out>
Out replace_copy_if(I first, I last, Out result, P pred, const T& new_value);
```

Each of these algorithms imposes two writability requirements on the Out parameter. This is due to the fact that T is allowed to be different than ValueType<I>; the algorithm writes values of both types through Out.

An alternative design substitutes the free template parameter T with the ValueType<I>, resulting in a slightly stronger requirement. This would also let us remove the IndirectlyCopyable requirement since it would then become redundant with Writable<T, Out>. Again, stronger requirements allow for more succinct algorithm specifications.

2.2.6 Fill

The fill and fill_n algorithms copy a value into each element in a specified range. The declarations and requirements of these algorithms vary from those found in the standard.

```
template<WeaklyIncrementable Out, typename T>
requires EqualityComparable<Out> && Writable<T, Out>
void fill(Out first, Out last, const T& value);
```

Unlike in the standard, fill does not require the template argument Out to be a ForwardIterator. The reason we relax this requirement is that we allow output iterators with equality comparison. We allow these iterators because they are a direct reflection of the actual requirements of the algorithm. This concept (output iterators with equality comparison) was missing from STL and subsequently the C++ standard. We should be able to use, for example, a back insertion iterator to fill a bounded queue.

There are exactly three algorithms in the STL with exactly these type requirements: fill, generate (§2.2.7), and iota in the numeric library. Over the course of this project, we had considered creating new concepts to express the combined requirements of these algorithms. In the end, it wasn't immediately obvious that the new concepts substantially improved the design or requirements, so we leave further investigation of these concepts as future work.

The requirements of fill_n are similar to copy_n.

```
template<WeaklyIncrementable Out, typename T>
requires Writable<T, Out>
Out fill_n(Out first, DistanceType<Out> n, const T& value);
```

2.2.7 Generate

The generate algorithm is closely related to the fill algorithm in terms of behavior and requirements.

```
template<WeaklyIncrementable Out, Function F>
requires EqualityComparable<Out> && Writable<ResultType<F>, Out>
F generate(Out first, Out last, F gen);

template<WeaklyIncrementable Out, Function F>
requires Writable<ResultType<F>, Out>
pair<Out, F> generate_n(Out first, DistanceType<Out> n, F gen);
```

The first overload has the many of the same requirements as fill (§2.2.6), except that the result of f is written to each iterator in the range [first, last). Because F is a nullary function (taking no arguments), we are allowed to write Function as the template argument type of F. Note that Function is a variadic concept; we can use it to write requirements for functions of any arity.

The signature of these functions varies from that found in the standard; we return the function f. Because F is allowed to modify non-local variables or its own internal state, we should allow the user to observe the state of the function after generating values, which we do by returning the function. In general, any algorithm taking a non-regular function should return that function.

2.2.8 Remove

The remove family of algorithms remove elements from a range in two ways. The in-place versions permute the elements of the range, dividing it into two parts: elements that have been retained and the remainder. The *_copy algorithms traverse the range, only copying those elements into the output that will be retained. The in-place declarations are:

```
template<ForwardIterator I, EqualityComparable<ValueType<I>> T>
requires Permutable<I>
I remove(I first, I last, const T& value);

template<ForwardIterator I, Predicate<ValueType<I>> P>
requires Permutable<I>
I remove_if(I first, I last, P pred);
```

The Permutable concept, which requires IndirectlyMovable<I, I>, allows the algorithm to exchange elements in a range using move. Permuting the elements of a range means that one element may be moved into the address of another. In the remove and unique algorithms, the move is "one-sided"; the moved-from object is left partially formed. In other algorithms such as shuffle, permutations are achieved by swapping elements.

The Permutable concept also requires its iterator argument to be a ForwardIterator. This means that the requirements given for the algorithm are redundant. Our convention is to state the strongest requirement on a template argument as its "type". We feel that better communicates the abstractions to the user.

Note that, for any result iterator i, the elements in the range [i, last) may be partially formed.

The remove_copy and remove_copy_if algorithms remove elements from a range by simply not copying the removed elements into the output.

```
template<InputIterator I, WeaklyIncrementable Out, EqualityComparable<ValueType<I>> T>
requires IndirectlyCopyable<I, Out>
Out remove_copy(I first, I last, Out result, const T& value);

template<InputIterator I, WeaklyIncrementable Out, Predicate<ValueType<I>> P>
requires IndirectlyCopyable<I, Out>
Out remove_copy_if(I first, I last, Out result, P pred);
```

Here, IndirectlyCopyable is required instead of Permutable because the original range is preserved by the operation.

2.2.9 Unique

The unique family of algorithms are virtually identical in their requirements to the remove family of algorithms. Both versions of unique rearrange the sequence so that no adjacent elements are equal (or equivalent). The unique_copy algorithms copy the elements of the original range into an output range so that the same property is satisfied. Their declarations are:

```
template<ForwardIterator I>
requires EqualityComparable<ValueType<I>> && Permutable<I>
I unique(I first, I last);

template<ForwardIterator I, Relation<ValueType<I>> R>
requires Permutable<I>
I unique(I first, I last, R comp);
// precondition: comp is an equivalence relation

template<InputIterator I, WeaklyIncrementable Out>
requires EqualityComparable<ValueType<I>> && IndirectlyCopyable<I, Out>
Out unique_copy(I first, I last, Out result);

template<InputIterator I, WeaklyIncrementable Out, Relation<ValueType<I>> R>
requires IndirectlyCopyable<I, Out>
Out unique_copy(I first, I last, Out result, R comp);
// precondition: comp is an equivalence relation
```

In addition to requiring R to be a Relation, these algorithms have a precondition that comp defines an *equivalence relation*.

The C++ standard phrases the requirements slightly differently, requiring copy assignment only if neither I nor Out is a ForwardIterator. The effect is to mandate additional specializations that would have the following signatures:

```
template<ForwardIterator I, WeaklyIncrementable Out>
requires EqualityComparable<ValueType<I>>
    && IndirectlyCopyable<I, Out>
Out unique_copy(I first, I last, Out result);

template<InputIterator I, ForwardIterator Out>
requires EqualityComparable<ValueType<I>>
    && IndirectlyCopyable<I, Out>
Out unique_copy(I first, I last, Out result);
```

The first specialization uses a second iterator instead of a temporary value, thus avoiding an extra copy. The second specialization reads the current value from the Out range instead of using a temporary or second iterator. Similar specializations are needed for the generalized unique_copy that is parameterized over a Relation argument.

The compiler will choose the most specialized overload based on the properties of the deduced type arguments.

2.2.10 Reverse and Rotate

The reverse and rotate algorithms also permute the sequence of elements in their given range.

```
template<BidirectionalIterator I>
requires Permutable<I>
void reverse(I first, I last);

template<ForwardIterator I>
requires Permutable<I>
I rotate(I first, I middle, I last);
```

Reversing the elements of a sequence requires a BidirectionalIterator while rotating the elements around a midpoint only requires a ForwardIterator.

The *_copy variants of these algorithms copy the results to an output range rather than modifying the sequence in-place.

```
template<BidirectionalIterator I, Incrementable Out>
requires IndirectlyCopyable<I, Out>
Out reverse_copy(I first, I last, Out result);

template<ForwardIterator I, Incrementable Out>
requires IndirectlyCopyable<I, Out>
Out rotate_copy(I first, I middle, I last, Out result);
```

2.2.11 Random Shuffle

There are three algorithms for shuffling a sequence of objects. The random_shuffle algorithms predate the more advanced C++11 random number facilities, which are used by shuffle. All of these algorithms randomly permute the given range.

```
template<RandomAccessIterator I>
requires Permutable<I>
void random_shuffle(I first, I last);

template<RandomAccessIterator I, RandomNumberGenerator<DistanceType<I>> Gen>
requires Permutable<I>
void random_shuffle(I first, I last, Gen&& rand);
```

A RandomNumberGenerator (§3.8) is a unary, non-regular Function defined over an Integral type. That is, it accepts an Integral argument, n, and returns a randomly selected value (of the same type) m where 0 <= m < n. All choices for m are equally likely.

```
template<RandomAccessIterator I, UniformRandomNumberGenerator Gen>
requires Permutable<I>
void shuffle(I first, I last, Gen&& rand);
```

A UniformRandomNumberGenerator (§3.8) is a nullary, non-regular Function that computes a pseudo-random sequence of uniformly distributed random UnsignedIntegral values in the interval [0, numeric_limits<ResultType<Gen>>::max()).

2.2.12 Partitions

The partition family of algorithms deals with ranges that are partitioned respect to some predicate. That is, there is a subrange of elements that all satisfy the predicate followed by a subrange of elements that does not. The is_partitioned returns true if the range is partitioned, and partition_point returns the iterator that divides elements satisfying the predicate from those that do not.

```
template<InputIterator I, Predicate<ValueType<I>> P>
bool is_partitioned(I first, I last, P pred);

template<ForwardIterator I, Predicate<ValueType<I>> P>
I partition_point(I first, I last, P pred);
```

The partition and stable_partition algorithms permute a range of elements with respect to a predicate so that they are partitioned.

```
template<ForwardIterator I, Predicate<ValueType<I>> P>
requires Permutable<I>
I partition(I first, I last, P pred);

template<ForwardIterator I, Predicate<ValueType<I>> P>
requires Permutable<I>
I stable_partition(I first, I last, P pred);
```

The stable_partition algorithm requires ForwardIterator whereas the standard requires BidirectionalIterator. A range can be partitioned, preserving the initial order of its elements, using ForwardIterators.

The partition_copy algorithm "splits" the input range by copying elements into one of two output ranges.

```
template<InputIterator I, Incrementable Out1, Incrementable Out2, Predicate<ValueType<I>> P>
requires IndirectlyCopyable<I, Out1>
    && IndirectlyCopyable<I, Out2>
pair<Out1, Out2> partition_copy(I first, I last,
                                Out1 out_true, Out2 out_false,
                                P pred);
```

2.3 Sorting Algorithms

The sorting algorithms are concerned with orderings. Every algorithm in this family either requires a total ordering of value types, or is parameterized over a strict weak ordering as a generalization of a total order. In this section, every Relation template parameter is required to be a *strict weak ordering* (§3.4.1).

2.3.1 The Sort Family

The sorting algorithms are concerned with the ordering of elements in a range with respect to an order. The is_sorted and is_sorted_until algorithms determine if a range is in sorted order, or the point at which a range is not sorted, respectively.

```
template<ForwardIterator I>
requires TotallyOrdered<ValueType<I>>
bool is_sorted(I first, I last);

template<ForwardIterator I, Relation<ValueType<I>> R>
bool is_sorted(I first, I last, R comp);

template<ForwardIterator I>
requires TotallyOrdered<ValueType<I>>
I is_sorted_until(I first, I last);

template<ForwardIterator I, Relation<ValueType<I>> R>
I is_sorted_until(I first, I last, R comp);
```

The TotallyOrdered concept requires its template argument to supply the inequality operators (<, >, <=. and >=), and that its encoded values are totally ordered with respect to those operators (§3.3). Technically, this is a stronger requirement than the standard. The standard phrases these requirements in terms of LessThanComparable, which only needs < and semantically relaxes its meaning to that of a strict weak ordering.

A specialization of these algorithms could be written using InputIterators instead of ForwardIterators, but only if ValueType<I> is Semiregular (i.e. it must be copyable). An implementation might look like this:

```
template<InputIterator I>
requires Semiregular<ValueType<I>> && TotallyOrdered<ValueType<I>>
I is_sorted_until(I first, I last)
{
  if (first != last) {
    ValueType<I> prev = *first;
    ++first;
    while (first != last) {
      if (*first < prev)
        return false;
      prev = *first;
      ++first;
    }
  }
  return first;
}
```

The algorithm differs in performance from that above by a constant factor related to the cost of $\mathcal{O}(n)$ copies. This is similar to the specializations mandated by the C++ standard for the unique_copy algorithms (C++ Standard, alg.unique).

We have opted not to replace the original specification of the algorithm with this seemingly more general version. It isn't actually more general; although the iterator requirement is relaxed, the Semiregular<ValueType<I>> requirement limits the algorithm to cases where the iterator's value type is copyable. You couldn't use this, for example, to determine if a sequence of unique_ptrs are sorted (indirectly) by their values.

The rest of the sorting algorithms order the elements in a sequence by permuting the sequence. The sort algorithm has two overloads, declared as:

```
template<ForwardIterator I>
requires Sortable<I>
void sort(I first, I last);

template<ForwardIterator I, Relation<ValueType<I>> R>
requires Sortable<I, R>
void sort(I first, I last, R comp);
```

A sequence can be efficiently sorted in $\mathcal{O}(n \log n)$ time using ForwardIterators. A conforming algorithm is given in *Elements of Programming* (Stepanov and McJones, 2009). This is weaker than the BidirectionalIterator requirement given in the standard.

The Sortable concept is related to the Permutable concept discussed in §3.6. It's also overloaded. The single-parameter concept is parameterized over a ForwardIterator I and requires:

- that I is Permutable, and
- that ValueType<I> is Semiregular and TotallyOrdered

The two-parameter version has similar requirements except that it is parameterized over a Relation, which generalizes the total ordering requirement.

Sortable requires the value type to be Semiregular because quicksort makes a copy of the pivot element to compare against other elements (Hoare, 1969). The algorithm requiring only ForwardIterators in EoP uses a temporary buffer. Although there are some algorithms that do not require data elements to be copied—insertion sort is one such algorithm—we enable a greater degree of freedom for implementations by allowing data to be copied.

Note that we could also have chosen to declare the first sort overload as:

```
template<Sortable I>
void sort(I first, I last);
```

replacing the template argument type of I with Sortable. We chose not to do this for a purely stylistic reason: the overloads of sort would not appear to have the same requirements.

The stable_sort and partial_sort algorithms have similar requirements.

```
template<ForwardIterator I>
requires Sortable<I>
void stable_sort(I first, I last);

template<ForwardIterator I, Relation<ValueType<I>> R>
requires Sortable<I, R>
void stable_sort(I first, I last, R comp);

template<RandomAccessIterator I>
requires Sortable<I>
void partial_sort(I first, I middle, I last);

template<RandomAccessIterator I, Relation<ValueType<I>> R>
requires Sortable<I, R>
void partial_sort(I first, I middle, I last, R comp);
```

The partial_sort algorithm requires I to model RandomAccessIterator (§3.5.3). Efficient execution of the algorithm requires the iterator to advance arbitrary steps in constant time.

The partial_sort_copy algorithm is somewhat more complex. It copies the n smallest elements in [first, last) into an output range. Most of the sorting is done on the output range, but the algorithm also compares values between the two ranges.

```
template<InputIterator I1, RandomAccessIterator I2>
requires TotallyOrdered<ValueType<I1>, ValueType<I2>>
    && IndirectlyCopyable<I1, I2>
    && Sortable<I2>
I2 partial_sort_copy(I1 first, I1 last, I2 result_first, I2 result_last);

template<InputIterator I1, RandomAccessIterator I2, Relation<ValueType<I>> R>
requires TotallyOrdered<ValueType<I1>, ValueType<I2>>
    && IndirectlyCopyable<I1, I2>
    && Sortable<I2, R>
I2 partial_sort_copy(I1 first, I1 last, I2 result_first, I2 result_last, R comp);
```

The TotallyOrdered requirement is a *cross-type ordering* (§3.3). Much like the cross-type EqualityComparable concept, it generalizes the requirements of a total ordering over two different, but related types. The IndirectlyCopyable concept is needed because values are copied from the input range into the output range. The Sortable requirement reflects the fact that all of the actual sorting occurs in the output range.

The nth_element algorithm is also in the sorting family. It partitions the input range in such a way that the first + nth element will be the n^{th} element in the total order, and all elements up to the nth position will not be greater than those after.

```
template<RandomAccessIterator I>
requires Sortable<I>
void nth_element(I first, I nth, I last);

template<RandomAccessIterator I, Relation<ValueType<I>> R>
requires Sortable<I, R>
void nth_element(I first, I nth, I last, R comp);
```

The requirements of nth_element should be familiar.

2.3.2 Binary Search

The binary search algorithms define a family of related operations on sorted data. All of these algorithms in this section have an implied precondition that their input ranges [first, last) are sorted either by the total ordering of their value types or with respect to the given (strict) weak ordering.

The fact the input sequence is sorted admits two operations: lower_bound and upper_bound. For any value of the iterator's value type, say x, these operations return iterators into the input range that partition it into a lower range where all values are less than x and an upper range where all values are greater than x. The declaration of these algorithms are:

```
template<ForwardIterator I, TotallyOrdered<ValueType<I>> T>
I lower_bound(I first, I last, const T& value);

template<ForwardIterator I, TotallyOrdered<ValueType<I>> T>
I upper_bound(I first, I last, const T& value);
```

These algorithms operate on ForwardIterators whose ValueTypes are totally ordered. Because the value type T is allowed to be different than the value type, we use the cross-type TotallyOrdered concept to ensure proper semantics for the comparison. As with most algorithms in the STL, lower_bound and upper_bound have overloads that generalize the operation over a weak order parameter.

```
template<ForwardIterator I, Relation<ValueType<I>> R>
I lower_bound(I first, I last, const ValueType<I>& value, R comp);
```

```
template<ForwardIterator I, Relation<ValueType<I>> R>
I upper_bound(I first, I last, const ValueType<I>& value, R comp);
```

In these algorithms, we have replaced the type of value with the ValueType<I>; it is no longer a template parameter. The motivation for this change is improved specificity. The lower and upper bound of a sorted sequence are defined with respect to a value of the same type. It should not be possible, for example, to find for the lower bound in a sequence of employee objects, the string "Pat Riley". Note that the declaration allows values to be converted to the ValueType so a Same type constraint is not needed.

There is an alternative declaration for these overloads that we could consider:

```
template<ForwardIterator I, TotallyOrdered T, Relation<ValueType<I>, T> R>
I lower_bound(I first, I last, const T& value, R comp);
```

Here, we are allow the type of value to vary as a template parameter, but we constrain it using a cross-type relation concept (§3.4.1). Among other things, this requires that T and ValueType<I> share a common type. The difference between this declaration and that given above is subtle.

In the first declaration, any needed conversions happen at the call site. If ValueType<I> is string, and the user calls the algorithm using a const char*, a conversion is required. In the second overload, no conversion is applied. Instead, the function relies on the fact that T and the ValueType share a common type and that the appropriate overloads are defined. It doesn't actually matter what the common type is.

```
template<ForwardIterator I, TotallyOrdered<ValueType<I>> T>
pair<I, I> equal_range(I first, I last, const T& value);

template<ForwardIterator I, Relation<ValueType<I>> R>
pair<I, I> equal_range(I first, I last, const ValueType<I>& value, R comp);

template<ForwardIterator I, TotallyOrdered<ValueType<I>> T>
bool binary_search(I first, I last, const T& value);

template<ForwardIterator I, Relation<ValueType<I>> R>
bool binary_search(I first, I last, const ValueType<I>& value, R comp);
```

The standard gives a different precondition for all binary search algorithms. The standard requires their input ranges to be partitioned with respect an inequality. For example, the standard gives precondition on lower_bound as, "the elements e of [first, last) are partitioned with respect to the expression: e < value." This guarantees that a lower bound can be found for the stated property. However, this precondition is not sufficient to admit an upper bound on the same data set. Consider the following program:

```
int a[] = {1, 0, 2, 3, 2, 4};
auto l = lower_bound(a, a + 6, 2); // returns a + 2;
auto u = upper_bound(a, a + 6, 2); // error: not partitioned
```

We can can find the lower bound of the set of values because the data is properly partitioned with respect to the expression *i < 2, for each *i in [a, a + n). Trying to find the upper bound results in an assertion, assuming that preconditions are asserted. The input is not partitioned with respect to the requirement given in the standard for upper_bound: !(2 < *i).

We see this as a generalization error in the standard. Lower and upper bounds are properties of a sorted sequence of elements. In fact, all of these algorithms are in the same family precisely because they share the precondition that the input is sorted. If a programmer needs to find where a sequence is partitioned with respect to some inequality, they should use the partition_point algorithm.

2.3.3 Merge

The merge algorithms combine two sorted ranges into a sorted output range. That the input ranges are sorted is a precondition to the algorithms.

```
template<InputIterator I1, InputIterator I2, Incrementable Out>
requires Mergeable<I1, I2, Out>Out merge(I1 first1, I1 last1, I2 first2, I2 last2, Out result);

template<InputIterator I1,
         InputIterator I2,
         Incrementable Out,
         Relation<ValueType<I1>, ValueType<I2>> R>
requires Mergeable<I1, I2, Out, R>Out merge(I1 first1, I1 last1, I2 first2, I2 last2, Out result, R comp);
```

The Mergeable concept states the requirements needed to implement these algorithms (§3.6). The iterators I1 and I2 must be IndirectlyCopyable to Out, and their value type must be cross-type TotallyOrdered. The second overload uses a version of Mergeable that defines these requirements with respect to a cross-type Relation instead of the total ordering of the iterator's value types.

The in-place version of the merge algorithm combines two sorted sub-ranges so that the entire range is sorted after completion.

```
template<ForwardIterator I>
requires Sortable<I>
void inplace_merge(I first, I middle, I last);

template<ForwardIterator I, Relation<ValueType<I>> R>
requires Sortable<I, R>
void inplace_merge(I first, I middle, I last, R comp);
```

Because the operation is applied in place, we require Sortable instead of Mergeable. Note that the standard incorrectly requires BidirectionalIterators for the inplace_merge algorithm. An equivalent algorithm exists for merging ranges in place using only ForwardIterators.

2.3.4 Set Operations

The set algorithms implement common set-theoretic algorithms over sorted sequences of elements. As with the merge and inplace_merge algorithms, the general precondition of all set algorithms is that their input ranges are sorted, either by the total ordering of their value types or with respect to a given (strict) weak order.

The includes algorithm is the only query in this family. It determines if one range is a subset of the other. Again, there are two overloads: one relying on the total ordering of the value type, the other generalized over a weak order.

```
template<InputIterator I1, InputIterator I2>
requires TotallyOrdered<ValueType<I1>, ValueType<I2>>
bool includes(I1 first1, I1 last1, I2 first2, I2 last2);

template<InputIterator I1,
         InputIterator I2,
         Relation<ValueType<I1>, ValueType<I2>> R>
bool includes(I1 first1, I1 last1, I2 first2, I2 last2, R comp);
```

The only requirements on the first overload (besides the implied preconditions) are that the iterator's value type is totally ordered (by the cross-type ordering concept). The second overload generalizes the ordering of a strict weak ordering, R, which is required to be a cross-type Relation.

The set_union, set_intersection, set_difference, and set_symmetric_difference algorithms define basic algebraic operations on set abstractions. Their declarations are:

```
template<InputIterator I1, InputIterator I2, WeaklyIncrementable Out>
requires Mergeable<I1, I2, Out>
Out set_union(I1 first1, I1 last1, I2 first2, I2 last2, Out result);

template<InputIterator I1, InputIterator I2, WeaklyIncrementable Out>
requires Mergeable<I1, I2, Out>
Out set_intersection(I1 first1, I1 last1, I2 first2, I2 last2, Out result);

template<InputIterator I1, InputIterator I2, WeaklyIncrementable Out>
requires Mergeable<I1, I2, Out>
Out set_difference(I1 first1, I1 last1, I2 first2, I2 last2, Out result);

template<InputIterator I1, InputIterator I2, WeaklyIncrementable Out>
requires Mergeable<I1, I2, Out>
Out set_symmetric_difference(I1 first1, I1 last1, I2 first2, I2 last2, Out result);
```

Each of these algorithms requires Mergeable<I1>I2Out. Not surprisingly, the same set of operations used to merge sorted sequences can be applied to compute the union, intersection, difference and symmetric difference of sets.

For each of the algorithms above, there is a corresponding generalization over a weak order. As with the previous set of algorithms, these simply require the generalized Mergeable concept. They are:

```
template<InputIterator I1,
         InputIterator I2,
         WeaklyIncrementable Out,
         Relation<ValueType<I1>, ValueType<I2>> R>
requires Mergeable<I1, I2, Out, R>
Out set_union(I1 first1, I1 last1, I2 first2, I2 last2, Out result, R comp);

template<InputIterator I1,
         InputIterator I2,
         WeaklyIncrementable Out,
         Relation<ValueType<I1>, ValueType<I2>> R>
requires Mergeable<I1, I2, Out, R>
Out set_intersection(I1 first1, I1 last1, I2 first2, I2 last2, Out result, R comp);

template<InputIterator I1,
         InputIterator I2,
         WeaklyIncrementable Out,
         Relation<ValueType<I1>, ValueType<I2>> R>
requires Mergeable<I1, I2, Out, R>
Out set_difference(I1 first1, I1 last1, I2 first2, I2 last2, Out result, R comp);

template<InputIterator I1,
         InputIterator I2,
         WeaklyIncrementable Out,
         Relation<ValueType<I1>, ValueType<I2>> R>
requires Mergeable<I1, I2, Out, R>
Out set_symmetric_difference(I1 first1, I1 last1, I2 first2, I2 last2, Out result, R comp);
```

2.3.5 Heap Operations

A *heap* is a tree with the property (called the *heap property* or *heap-order* property) that the value at every node is not less than the values of its children. More specifically, this is called a max-heap; a min-heap has the property that a node is not greater than its children. The

algorithms in this family describe operations on *binary heaps*, which are defined over complete binary trees. A complete binary tree can be implemented using a random-access sequence.

The is_heap and is_heap_until algorithms query the heap property of a range of values. They ensure that every element is not less than its children. The algorithms are also generalized for any weak order. The declarations of these functions are:

```
template<ForwardIterator I>
requires TotallyOrdered<ValueType<I>>
bool is_heap(I first, I last);

template<ForwardIterator I, Relation<ValueType<I>> R>
bool is_heap(I first, I last, R comp);

template<ForwardIterator I>
requires TotallyOrdered<ValueType<I>>
I is_heap_until(I first, I last);

template<ForwardIterator I, Relation<ValueType<I>> R>
I is_heap_until(I first, I last, R comp);
```

The requirements are straightforward. Checking the heap property of a sequence requires a ForwardIterator and that its value type is TotallyOrdered. The push_heap and pop_heap algorithms are used to update a heap-ordered sequence after an object has been inserted or removed.

```
template<RandomAccessIterator I>
requires Sortable<I>
void push_heap(I first, I last);

template<RandomAccessIterator I, Relation<ValueType<I>> R>
requires Sortable<I, R>
void push_heap(I first, I last, R comp);

template<RandomAccessIterator I>
requires Sortable<I>
void pop_heap(I first, I last);

template<RandomAccessIterator I, Relation<ValueType<I>> R>
requires Sortable<I, R>
void pop_heap(I first, I last, R comp);
```

These algorithms require RandomAccessIterators. During insertion or removal, parts of the sequence are compared and swapped (i.e., sorted), which leads to the Sortable requirement.

The make_heap algorithm has similar requirements because it performs many of the same operations found in push_heap and pop_heap.

```
template<RandomAccessIterator I>
requires Sortable<I>
void make_heap(I first, I last);

template<RandomAccessIterator I, Relation<ValueType<I>> R>
requires Sortable<I, R>
void make_heap(I first, I last, R comp);
```

Finally, sort_heap takes a heap-ordered sequence and produces a sorted sequence using either the total ordering of the value type or the weak order comp.

```
template<RandomAccessIterator I>
requires Sortable<I>
void sort_heap(I first, I last);
```

```
template<RandomAccessIterator I, Relation<ValueType<I>> R>
requires Sortable<I, R>
void sort_heap(I first, I last, R comp);
```

Technically, the arity of the tree used in these algorithms is unspecified. We can just as easily implement these algorithms for complete ternary or quaternary trees. The requirements would be the same in any case.

2.3.6 Minimum and Maximum

There are 20 algorithms for finding the min and max. Arguably, the most commonly used overloads of min are:

```
template<TotallyOrdered T>
const T& min(const T& a, const T& b);

template<typename T, Relation<T> R>
const T& min(const T& a, const T& b, R comp);
```

The algorithm returns the least of the two values, either according to their total ordering or with respect to the weak order comp. Note that T is unconstrained in the second overload. Since a and b are passed by reference and the comparison is encapsulated by the comp parameter, we cannot assign any stronger requirements to T.

Another two overloads of min take an initializer_list containing a sequence of values.

```
template<TotallyOrdered T>
const T& min(initializer_list<T> t);
// different than standard

template<typename T, Relation<T> R>
const T& min(initializer_list<T> t, R comp);
// different than standard
```

The type requirements are identical to those in the previously described versions of min. However, the declarations differ from the standard. We return a const reference to the minimum value. The reason for this (and also the reason why we don't require T to be Semiregular), is that the initializer list already refers to non-local data. The values in t are not copied into the stack frame of the function.

There are four corresponding overloads of max with requirements identical to those of min.

```
template<TotallyOrdered T>
const T& max(const T& a, const T& b);

template<typename T, Relation<T> R>
const T& max(const T& a, const T& b, R comp);

template<TotallyOrdered T>
const T& max(initializer_list<T> t);

template<typename T, Relation<T> R>
const T& max(initializer_list<T> t, R comp);
```

Again, we have changed the result of the initializer_list algorithms to return a const reference.

There are also four corresponding overloads of minmax, which find the min and max simultaneously. Again the requirements of these algorithms are identical to those of min and max.

```
template<TotallyOrdered T>
pair<const T&, const T&> minmax(const T& a, const T& b);
```

```
template<typename T, Relation<T> R>
pair<const T&, const T&> minmax(const T& a, const T& b, R comp);

template<TotallyOrdered T>
pair<const T&, const T&> minmax(initializer_list<T> t);

template<typename T, Relation<T> R>
pair<const T&, const T&> minmax(initializer_list<T> t, R comp);
```

Curiously, there are no overloads of min, max, or minmax that take and return non-const references. This seems to be an omission from the standard.

The min_element, max_element, and minmax_element extend the computation of min, max, and minmax to iterator ranges.

```
template<ForwardIterator I>
requires TotallyOrdered<ValueType<I>>
I min_element(I first, I last);

template<ForwardIterator I, Relation<ValueType<I>> R>
I min_element(I first, I last, R comp);

template<ForwardIterator I>
requires TotallyOrdered<ValueType<I>>
I max_element(I first, I last);

template<ForwardIterator I, Relation<ValueType<I>> R>
I max_element(I first, I last, R comp);

template<ForwardIterator I>
requires TotallyOrdered<ValueType<I>>
pair<I, I> minmax_element(I first, I last);

template<ForwardIterator I, Relation<ValueType<I>> R>
pair<I, I> minmax_element(I first, I last, R comp);
```

These algorithms all operate on ForwardIterators. As with is_sorted, and is_sorted_until, we could conceivably relax the requirements to InputIterator, at the expense of extra copies and an additional requirement that ValueType<I> is Semiregular.

2.3.7 Lexicographical Comparison

Two sequences can be lexicographically compared either by the total ordering of their value types or with respect to a weak order. These features are implemented by the two overloads of the lexicographical_compare algorithm.

```
template<InputIterator I1, InputIterator I2>
requires TotallyOrdered<ValueType<I1>, ValueType<I2>>
bool lexicographical_compare(I1 first1, I1 last1, I2 first2, I2 last2);

template<InputIterator I1,
         InputIterator I2,
         Relation<ValueType<I1>, ValueType<I2>> R>
bool lexicographical_compare(I1 first1, I1 last1, I2 first2, I2 last2, R comp);
```

The first overload requires only that the value types of the iterators are TotallyOrdered as specified by the cross-type ordering concept. The second overload is defined in terms of a cross-type Relation defined over the value types of I1 and I2.

2.3.8 Permutation Generators

The permutation generators permute a sequence according to a total or weak order to generate all permutations of the original input. The next_permutation and prev_permutation generate the next or, respectively, the previous permutation according to the order imposed on permutations by the corresponding lexicographical orderings. These are essentially sorting algorithms that operate on BidirectionalIterators. Their type requirements are fairly straightforward.

```
template<BidirectionalIterator I>
requires Sortable<I>
bool next_permutation(I first, I last);

template<BidirectionalIterator I, Relation<ValueType<I>> R>
requires Sortable<I, R>
bool next_permutation(I first, I last, R comp)

template<BidirectionalIterator I>
requires Sortable<I>
bool prev_permutation(I first, I last);

template<BidirectionalIterator I, Relation<ValueType<I>> R>
requires Sortable<I, R>
bool prev_permutation(I first, I last, R comp)
```

3 Concepts

In this section, we detail the definition of concepts used to constrain the algorithms described in §2, and we describe the syntax used to construct those concepts. We begin by introducing some fundamental ideas.

3.1 Preliminaries

The concepts in this report reinforce the ideas about computer programming presented in the book, *Elements of Programming* (Stepanov and McJones, 2009). At the heart of these ideas is the notion that we should be able to reason about computer programs (e.g. behavior, correctness, performance, etc.). To do so, we require a basic set of guarantees that the elements of a computer program behave in expected and regular ways. In this section, we describe the basis of these guarantees, which are rooted in the notions of values, objects, equality, and regularity.

All computer programs represent abstract entities (e.g., numbers, colors, books, etc.) in memory as sequences of 1's and 0's called *data*. Obviously, writing programs as manipulations of binary data is tedious, error-prone, and does not scale. Programming languages use *types* to correlate data with abstract entities. A *value type* specifies an interpretation of data as an abstract entity. In C++, a value type is an unqualified (i.e. neither const nor volatile), non-reference type such as int or string; value types describe the properties and behaviors of an entity. A *value* (of a type) is a particular instance of a datum and its interpretation. For example:

- The C++ type int specifies the computer encoding of, say, 32-bit, two's complement integer data in big-endian format. A sequence of 32 bits, all 0, is interpreted as the int value 0.
- A template specialization rgb_color<4> represents 4-bit RGB colors in a 12-bit string with 4 bits for each of three color components (red, green, and blue). The color or value red is represented by the sequence of bits, 1111 0000 0000.

Not all representations can be interpreted as values. A datum is *well formed* if its representation can be interpreted as an entity. For example, all sequences of 32 bits are well formed integers, as are all 12-bit sequences of rgb_color. On the other hand, the sequence of bits encoding a floating point NaN is not well formed since it does not represent any real number (it is literally "Not a Number"). Data that are not well formed are *ill-formed*. Similarly, a rational number with 0 in the denominator is ill-formed.

An *object* represents a value in memory. That is, an object holds a value or set of values called its *state*. An *object type* describes how a value type is stored and modified within computer memory. An object type adapts the properties of a *single* object's value type in different ways:

- An object's representation may be padded to align it with an address boundary; rgb_color<4> is almost certainly padded to 16 bits from its original 12, possibly more.
- An object's state may be made immutable. A *constant* object is constructed by declaring const-qualified variable. A constexpr object further restricts the declared object by ensuring that it only exists at compile time.
- An object's state may be changed by an external process, thread, or device. A *volatile object* is constructed using the volatile qualifier.

A *reference type* defines an alias to an object. Aliases can refer to objects in memory (lvalue references) or temporaries and literals (rvalue references). Like objects, references may also be qualified. A const locally restricts write access to the referenced object although the object itself may not be a constant. Reference types are generally indistinguishable from object types, unless a program cares to differentiate. For example, overload resolution and template specialization can distinguish between reference and non-reference types, and the is_reference (C++ Standard, meta.unary.comp) type trait can be used to distinguish programmatically.

3.1.1 Equality

Reasoning about computer programs is facilitated by *equational reasoning*, which allows us to substitute equals for equals. As such, equational reasoning depends on a meaningful definition of the *equality* of values and the expressions that compute them.

If two values represent the same entity, then they are *equal*. If two values have the same representation, then they are *representationally equal*. Note that there may be multiple representations for the same entity. For example, a rational type may multiple representations for the entity 1/2, such as 4/8 and 3/6.

Two objects are equal if they have equal values and are said to be *copies*. Values are copied between objects using a *copy constructor* or a *copy assignment* operator—the prerequisites of any type implementing *copy semantics*. These are special member functions that may be overloaded to propagate the properties of a representation that are interpreted as its value. For example, copying a vector will copy the contained elements, but not necessarily all of the allocated memory (its capacity). If no user-defined copy constructor or assignment operator is defined (or deleted), the compiler will generate an *implicit* default copy constructor and assignment operator which copy objects member-wise. Regardless of whether the copy constructor is user-defined or implicit, the value resulting from a copy must be equal to the original, and the original object is unmodified by the operation. We take this as the basis for describing equality.

Actual specifications of equality fall into one of three categories:

1. The class has a user-defined equality operator.
2. The class has a user-defined copy constructor but no equality operator.
3. The class has an implicit copy constructor, and no equality operator.

In the first case, the onus of guaranteeing the equality of copies is entirely that of the user. In other words, it would be appropriate to annotate the copy constructor such types with the postcondition that the newly constructed value is equal (==) to its argument. In the second case, equality can only be assumed. There is no == operator to verify that copies are equal. We expect this to be an uncommon case. In the last case, copied objects are guaranteed to have equal members because each member variable is a copy of an original member value. ‘ User-defined == operators are used throughout the language and standard library to determine if two values represent the same entity. That is, the operator is used to compare for “true” equality, not simply representational or member-wise equality. For example, the floating point values -0.0 and 0.0 have different representations, but -0.0 == 0.0 is true because they both represent the same abstract entity: zero. Similar abstractions of equality are derived for every type in the C++ standard library. Some examples include:

- Two tuples are equal if and only if they have the same number of sub-objects and all of their sub-objects compare equal using ==.
- Two vectors are equal if and only if they have the same size and contain the same elements as if compared using the equal algorithm (which compares using ==).
- Two complex numbers are equal if and only if they have equal real and imaginary components.

Each of these definitions of equality is associated with some specification. This is closely related to the description of value semantics described by Lakos (2007). There, the meaning of equality is based on a specification of salient attributes: a subset of properties that describe the value of a user-defined type.

Note that the interpretation of equality for implicitly copied types having no user-defined equality operator is not guaranteed to compare for “true” equality. It is, however, sufficient to guarantee that we can reason about programs when such copies are used.

Copy semantics are not the only means by which values are transferred between objects. A *move* transfers a value from one object to another, leaving the original object in a partially formed state. The value of the “receiving” object is equal to the original before it was moved. The syntax for move construction and assignment is similar to that of copy construction and assignment, except that the move function is used to indicate the different behavior.

```
T a = b;       // Copy construction
T a = move(b); // Move construction

a = b;         // Copy assignment
a = move(b);   // Move assignment
```

A type that supports move construction and assignment is said to implement *move semantics*.

Curiously, the move function does not actually *do* anything. It only helps select the most efficient method of transferring the value of b into a. For a large set of types the most efficient method of moving values may actually be to copy them. This is certainly true for all built-in scalar types (C++ Standard, basic.types/9) and probably also all trivial class types (C++ Standard, class/6).

All copyable types inherently implement move semantics. Unless otherwise specified, the “default” behavior of move operations is actually copying. However, a user-defined type may define a *move constructor* and *move assignment operator* to optimize the transfer. For example, the value of a vector can (very) efficiently moved by simply swapping its underlying pointers with that the receiving object. Additionally, the compiler can implicitly generate member-wise move operations in some cases. The behavior of these operations preserve the semantics described above.

There are many useful types that implement move semantics but not copy semantics. These non-copyable types often represent *resources*. For example, a non-nullptr value of a unique_ptr cannot be shared by another unique_ptr. However, it is often convenient to define equality for such types, even though they do not compare for "true" equality. Without this feature, we would not be able to use unique_ptrs with, say, unordered_maps.

To help write axioms involving language-defined, user-defined, or assumed equality, we introduce a built-in equivalence relation, eq. It has the following definition:

```
template<typename T>
bool eq(const T& a, const T& b)
{
  // Returns true if and only if a and b are equal according to the
  // strongest available interpretation of equality for type T.
}
```

The purpose of this predicate is to provide a kind of universal mechanism for equating objects when no == operator is available and without specifying the mechanism by which equality is defined. In essence, the comparison invokes the strongest definition of equality available for T. The eq predicate is not intended to be evaluated in programs; it is only useful in the specification of axioms.

3.1.2 Expressions

An *expression* is a sequence of operators and operands that specifies a computation. Expressions may result in a value and may have side effects. Exactly how the value is returned depends on the definitions of the operators applied. A value may be returned in an object (either copied or moved, or even a constexpr object), or as a reference (lvalue, rvalue, possibly const-qualified). Note that void expressions do not result in values.

An expression is *equality preserving* if, given equal inputs, the expression results in equal outputs. The EoP book uses the term *regular function* to describe functions that satisfy this property. Using the eq predicate, we might write this for some unary function f as follows. For any object x, the expression f(x) is equality preserving if, for any a and b with the same type as x:

```
eq(a, b) => eq(f(a), f(b))
```

In other words, if equal arguments always yield equal results, then f is equality preserving.

An expression that is generally equality preserving may not be for some specific representations. For example, the expression a / b is equality preserving for floating point types, except when b is -0.0 or 0.0. That is: 0.0 == -0.0 => 1/0.0 == 1/-0.0 is false. This does not contradict the general claim that division is an equality preserving expression; there is a general and well-known prohibition on dividing by 0. Any *generic* algorithm that divides by 0 invites undefined behavior, unless it is *specifically* designed to operate on IEEE 754 floating point types. Traditionally, exceptional cases like this are guarded by preconditions, signals, or exceptions.

In concepts, equality preserving expressions are the norm; writing this specification for every required syntactic expression would be tedious and repetitive. As such, we assume that all required expressions are equality preserving unless otherwise stated. To indicate that an expression is *not* equality preserving, we use the not_equality_preserving predicate. For example, in Function, we have the following axiom:

```
not_equality_preserving(f(args...));
```

This means that the results of applications of f may be different when evaluated over equal arguments. This does not say that f must not preserve equality, only that it is not required to.

We can strengthen not_equality_preserving requirements in subsequent concepts if needed by using the corresponding equality_preserving predicate. For example, in RegularFunction (which is a semantic refinement of Function), we have:

```
equality_preserving(f(args...));
```

This indicates that, for any equal sequence of arguments, the results of calling f will be the same.

By default, we assume that all required expressions are equality preserving unless stated otherwise. There are only a few required operations used in the STL that are not required to be equality preserving.

3.1.3 Type Functions

A *type function* is a compile-time function where at least one argument or the result is a type. For example, the following are built-in type functions:

```
sizeof(T)     // takes a type argument, returns a constexpr size_t value
alignof(T)    // takes a type argument, returns a constexpr size_t value
typeid(T)     // takes a type argument, returns an RTTI object
decltype(x)   // takes an expression argument, returns a type
```

A template alias is another kind of type function. Template aliases return types. For example, we can create a type function that returns the target type of a pointer:

```
template<typename T>
using PointeeType = remove_pointer<T>::type;

PointeeType<int*> // aliases int
PointeeType<int**> // aliases int*
```

The fact that we can encapsulate a template metaprogram using a template alias promotes abstraction and dramatically improves the ways in which we refer to types in generic algorithms. We have used this principle to completely bury the iterator_traits class; it is not used in the declaration of any algorithm in the previous sections. Not only that, we have, in template aliases, a mechanism for avoiding a profusion of ::'s in algorithm specifications. None of the algorithms described in Sections §2 contain a single instance of the scope resolution operator. We think this is a big improvement over the requirements written for C++0x (C++ Standards Committee, 2009).

Type functions can also take integral constants as values. For example, we can simplify the access of tuple element types using the following alias:

```
template<int N, typename... Args>
using Tuple_element = tuple_element<N, Args...>::type;

using First_type = Tuple_element<0, char, short, int, long>;  // aliases char
```

This alias is purely for exposition; it is not used in this specification of the STL or its concepts.

Concepts are a kind of *type predicate* that determine whether their type (or non-type) arguments satisfy a set of requirements. A concept is true if its arguments satisfy its requirements and false otherwise. For example, the concept Same is true if its arguments are exactly the same type. Same<int, int> is true, and Same<int, char> is false.

As with the algorithms presented in the previous sections, concepts are also grouped into related families. These are:

- Language concepts (§3.2)
- Foundational concepts (§3.3)
- Functions (§3.4)
- Iterators (§3.5)
- Rearrangements (§3.6)

- Random numbers (§3.8)

The following sections describe these families and their concepts in detail.

3.2 Language Concepts

We define seven concepts relating directly to language features: four that describe relationships between types and three that classify fundamental types. We imagine that these concepts will be intrinsic (i.e. , built into the compiler), although their truth values can be evaluated in C++0x using type traits. The STL requires the following language concepts:

```
// Type relations:
concept Same<typename T, typename U>
concept Derived<typename T, typename U>
concept Convertible<typename T, typename U>
concept Common<typename T, typename U>

// Type classifications:
concept Integral<typename T>
concept SignedIntegral<typename T>
concept UnsignedIntegral<typename T>
```

3.2.1 Type Relations

A concept definition introduces the name of a concept and its parameters. To define a concept, we specify the requirements for it to be true (valid):

```
concept Same<typename T, typename U> = // could be defined variadically
  is_same<T, U>::value;
```

The concept Same is described in terms of the standard-library type predicate is_same. The value of a concept (true or false) is equal to the predicate on the right-hand side of the =. We give the definition in terms of a type trait for exposition only. This concept will most likely be built in.

By "could be defined variadically," we mean that we could define concept Same<typename... Args>. However, we did not need that generalization for the STL algorithms, so we postponed the decision to define it more generally. We don't generalize beyond the need of the STL. That is—without prejudice—left as future work.

For types X and Y, Same<X,Y> is true if X and Y denote exactly the same type after elimination of aliases. For example:

```
using Pi = int*;
using I = int;
Same<Pi,I*> // is true
```

The Derived concept returns true if the first template argument is derived from the second.

```
concept Derived<typename T, typename U> =
  is_base_of<U, T>::value;
```

This concept expresses the requirement that T is derived from U; that is, that U is a base class of T. For example:

```
class B { };
class D : B { };

Derived<D, B> // is true: D is derived from B
```

```
Derived<B, D> // is false: B is base class of D
Derived<B, B> // is true: a class is derived from itself
```

The last case is true because Derived defines a (non-strict) subtype relation on inheritance hierarchies.

The implicit conversion between one type and another is tested using the Convertible concept, which is defined as:

```
concept Convertible<typename T, typename U> = is_convertible<T, U>::value;
```

This concept expresses the requirement that a type T can be implicitly converted to a U. T is converted to U if an expression of type T can be returned from a function with a result type U (C++ Standard, meta.rel). Some examples are:

```
Convertible<double,int>             // is true: int i = 2.7; (ok)
Convertible<double,complex<double>> // is true: complex<double> d = 3.14; (ok)
Convertible<complex<double>,double> // is false: double d = complex<double>2,3 (error)
Convertible<int,int>                // is true: a type is convertible to itself
Convertible<Derived, Base>          // is true: derived types can be converted to base types
```

Note that the meaning of Convertible is completely defined in terms of C++0x standard constructs.

We say that two types T and U are *unambiguously convertible* if there exists a unique, *explicit* conversion from T to U, or from U to T, but not both. This is trivially true when T and U are the same type since no conversion would be required.

If T and U can both be unambiguously converted to a third type, C, we say that T and U share a *common type*, C. Note that C could be the same as T, or U, or it could be a different type. This notion is encapsulated by the CommonType alias and the Common concept.

```
template<typename T, typename U>
using CommonType = common_type<T, U>::type;

concept Common<typename T, typename U> =
  requires {
    CommonType<T, U>;

    axiom (T t1, T t2, U u1, U u2) {
      using C = CommonType<T, U>;
      eq(t1, t2) <=> eq(C{t1}, C{t2});
      eq(u1, u2) <=> eq(C{u1}, C{u2});
    }
  }
```

Common and CommonType are essential for preserving the mathematical soundness of relations and operations on heterogeneous arguments (see §2.1.3). The CommonType alias refers to the common type of its template arguments, if it exists (in this report, CamelCase identifiers ending with "Type" denote type functions). We define the alias in terms of the standard common_type trait (C++ Standard, meta.trans.other/3), which is conventionally implemented in terms of the conditional operator, ?: (C++ Standard, expr.cond). The Common concept is true if its template arguments share a common type.

The Common concept introduces requirements differently than those above. A set of requirements are introduced by the identifier requires. We distinguish between

- requirements, which must be checked by the compiler, and
- axioms, which express semantics and must not be checked by the compiler.

The non-axiom parts of a requirement express the syntax required by a concept. They are statically checked by the compiler. The only syntactic requirement of the Common concept is:

```
CommonType<T, U>;
```

This requires the CommonType alias to be well-formed for the template arguments T and U. This is, in some respects, similar to the associated type requirements in C++0x, except that we do not allow default types to be assigned C++ Standards Committee (2009).

In order for the requirement to be satisfied, the alias must not result in a lookup failure. If the alias does result in a lookup failure, then requirement is not satisfied and the concept check will fail. For example, if T and U are string and int respectively, the requirement will fail: there is no explicit conversion from a string to an int or int to a string. This means that the concept Common<string, int> will be false. However, if T and U are int and double, the concept will be true since CommonType<int, double> will result in a valid type name (double).

Axioms express the semantics of a concept's required syntax; they are assumed to be true and must not be checked by the compiler beyond conformance to the C++ syntax. Any additional checking is beyond the scope of the compiler's translation requirements. The compiler must not generate code that evaluates axioms as preconditions, either. This could lead to program errors if the evaluated assertions have side effects. For example, asserting that distance(first, last) > 1 when the type of those iterators is istream_iterator will consume the first element of the range, causing the assertion to pass, but the algorithm to have undefined behavior.

An axiom block is introduced by the axiom identifier and may be followed by a list of objects used to express the meaning of required operations. Here, we introduce a number of objects of type T and U. The first line of the axiom is type alias; creating an alias for the common type of T and U makes the remaining axioms easier to write. The following statements:

```
eq(t1, t2) <=> eq(C{t1}, C{t2});
eq(u1, u2) <=> eq(C{u1}, C{u2});
```

State equations describing the properties of common types. The <=> operator is the biconditional operator. It states the logical equivalence of its arguments. We give a more thorough description of the operator in Appendix A. The Common concept guarantees that, if T and U share a common type C then

- equal values of type T will be converted to equal values of type C, and
- constructing equal values of type C implies that their original arguments (t1 and t2) are equal,
- and the same properties hold for U and C as well.

In total, these properties guarantee that values are preserved by conversion to the common type. This property is important when describing the semantics of cross-type relations such as EqualityComparable (§3.3), TotallyOrdered (secrefsec:foundation), Relation (§3.4.1), and BinaryOperation (§3.4.2). Note that we use the eq operation since there are no syntactic requirements for the == operator.

Both the Common concept and the CommonType alias could be defined variadically. After all, the common_type trait is defined that way (C++ Standard, meta.trans.other/3). However, this specification of the STL did not require evaluating the common type of more than two types. If a variadic CommonType alias is needed, then its definition could be easily amended.

As a relation on types, the Common concept is both reflexive (Common<T, T>) is true) and symmetric (Common<T, U> is the same as Common<U, T>). It is not transitive, so it is not an equivalence relation.

From a mathematical perspective, Common<T, U> is true if the values represented by the types T and U can be embedded in the same theoretical mathematical universe. For example, the values of int and double can be embedded in the extended real line, $\mathbb{R}$ (or the extended real number line $\overline{\mathbb{R}}$ if inf and -inf are considered), and so they share a common type. The Common concept determines the type that appropriately represents that abstract universe. For example:

```
Common<int, double>      // is true
CommonType<int, double>  // aliases double
```

Some other embeddings leading to common types are:

```
CommonType<char, long>           // aliases long
CommonType<int, unsigned int>    // aliases unsigned int
CommonType<float, long double>   // aliases long double
CommonType<int, float>           // aliases float
CommonType<const char*, string>  // aliases string
CommonType<int, int>             // aliases int

// But...
Common<employee, string>         // is probably false
```

The common type of built-in numeric types follows the C++ promotion rules. These rules promote built-in types to those capable of representing values with greater precision. There are obvious limitations. For example, adding a 64-bit long long and a 32-bit float will "promote" the long long type to the smaller float even though there are many possible long long values that cannot be represented by a float. The conversion is lossy.

Promotion rules for signed and unsigned integral types are also defined. For example, adding signed char and unsigned short will promote both values to int. This process is obviously limited by the largest integral type supported. Adding -1ll and 1ull results in overflow because there is no larger integer type that can accommodate the additional information. Unfortunately, we have inherited these conversion features from the C typing rules (C++ Standard, conv) and seem to be stuck with them for the time being.

C-strings and std::strings of the same character type can be embedded in a universe of character sequences. However, an employee class probably does not share a common type with string because there is no mathematical universe that embeds the values of both employees (people) and strings (not people). For the purpose of examples throughout this report, we assume that the stereotypical employee class does *not* share a common type with string.

This is an important observation because it may be common practice for programmers to overload comparison operators for non-interoperable types (e.g. employees and strings). For example, the following program might be used to find a name in a list of employees:

```
vector<employee> v;
// ...
auto r = find(v.begin(),v.end(),"Alice Smith");
```

In our design, the program will be invalid because const char* does not share a common type with employee. There is no unique sequence of conversions that can be applied to const char* that results in an employee object. This is the case even when there is an overloaded == operator taking an employee and a const char* arguments. The reason for this is that we cannot express a precise, mathematically sound meaning of equality for unrelated types. In fact the semantics of equality for different types (that share a common type) is defined in terms of the common type (see the cross-type EqualityComparable concept §3.3 for a more detailed explanation).

The CommonType facility is not "closed". Although it is initially defined only for built-in types, it is possible to extend the CommonType facility in two ways:

1. define an unambiguous, implicit conversion from one type to another, or
2. explicitly specialize the common_type trait.

The standard string class uses the first method. It defines a non-explicit constructor taking a C-string (const char*). This defines an unambiguous conversion. This happens to be the exact requirement for extending the domain of the conditional operator to the different types. However, simply providing the implicit conversion may not be sufficient for interoperability. The string

class provides overloads of its relational and concatenation operators for C-strings. That means that this program:

```
vector<string> v;
// ...
auto r = find(v.begin(),v.end(),"Alice Smith");
```

is valid. string and const char* share a common type (by virtue of an unambiguous, implicit conversion). If the string class only provided the conversion, the algorithm could convert the C-string "Alice Smith" to a string for each comparison; this is probably not efficient. Fortunately, the string class overloads operators for C-strings, which means that no conversions are actually used within the algorithm. In other words, the usual lookup rules for overloads are applied.

The other method of extending the CommonType facility is to specialize the common_type trait. The duration class in the chrono library (C++ Standard, chrono.duration) does this. All durations types, regardless of their underlying value type and ratio (e.g. millisecond, microsecond), are embedded in a universe of duration values. For any two duration types, there is a third that can represent both values. Suppose, for example, we have an application that (for some reason) accumulates time in 3/5ths and 2/7ths second increments. We can still compare these values.

```
using D1 = duration<int, ratio<3, 5>>;
using D2 = duration<long, ratio<2, 7>>;

D1 a{3};
D2 b{2};
assert(a > b); // it's valid!
```

The values of a and b can be compared because D1 and D2 share a common type, duration<long, ratio<1, 35>). As with implicit conversions, specializing CommonType is not sufficient to guarantee interoperability. The equality, inequality, and arithmetic operations of the duration class template are provided for specializations with different arguments in order to ensure the interoperability of all duration types.

Note that duration only defines an implicit conversion for a subset of compatible types. You can't assign, for example, a = b using the values from the previous listing. That's an error. This means that the expression true ? a : b will also result in an error, which is why specialization is needed to extend the CommonType function.

3.2.2 Type Classifications

Type classifications describe sets of fundamental C++ types.

```
concept Integral<typename T> = is_integral<T>::value;
concept SignedIntegral<typename T> = is_signed<T>::value;
concept UnsignedIntegral<typename T> = is_unsigned<T>::value;
```

These concepts express requirements that a concept parameter T is of integral type, signed integral type, or unsigned integral type (as defined in the C++11 standard (C++ Standards Committee, 2011). These concepts are fully defined by the predicates on the right of the =.

These concepts are not "open" in the sense that users cannot construct new models. They describe a closed set of types whose properties are formalized in the language standard. It is certainly possible, even reasonable, to define concepts describing classifications of other kinds of fundamental types, e.g. floating point types, pointers, references, etc. However, because we didn't need these concepts to constrain the STL algorithms, we did not define them. We leave this as future work.

We can define concepts as simple (constant) Boolean expressions. This is useful when building constraints predicated on non-syntactic and non-semantics properties of types. For example:

```
concept Small<typename T> = sizeof(T) <= CACHE_LINE_SIZE;
```

Such a concept might be useful when optimizing generic data structures for improved cache performance.

Note that writing Integral<const int> is equivalent to writing Integral<int>. Similarly, Integral<T&> == Integral<T>. These concepts evaluate the *value type* of their arguments—not the object type. This property is actually encoded as a requirement in the C++11 standard (C++ Standard, meta.unary.cat). In general, we design concepts to query properties of value types, not object types. This is not true of all concepts. In particular Same, Common, and Convertible are sensitive to differences in object type.

3.3 Foundational Concepts

A concept is foundational if it forms the basis of a style of programming or is needed to write programs in that style. Our design includes four foundational concepts (and two overloads) that describe the basis of the value-oriented programming style on which the STL is based. The purpose of these concepts is to establish a foundation for equational reasoning in programs. The foundational concepts are:

```
concept EqualityComparable<typename T>
concept EqualityComparable<EqualityComparable T1, EqualityComparable T2>

concept Semiregular<typename T>
concept Regular<Semiregular T>

concept TotallyOrdered<EqualityComparable T>
concept TotallyOrdered<TotallyOrdered T1, TotallyOrdered T2>
```

We consider these concepts so fundamental that they should be intrinsics (i.e., built-in concepts). We don't see any gain in letting a programmer define different versions of them. This is a decision aimed at keeping the conceptional framework sane, rather than a performance consideration (though it will help with compiler performance, also).

The ability to compare objects for equality is described by the EqualityComparable concept.

```
concept EqualityComparable<typename T> =
  requires (T a, T b, T c)
    bool { a == b };
    bool { a != b };
    axiom { a == b <=> eq(a, b); }
    axiom {
      a == a;
      a == b => b == a;
      a == b && b == c => a == c;
    }
    axiom { a != b <=> !(a == b); }
};
```

The meaning of EqualityComparable is straightforward, but the way we state it involves notation that is novel in this context. Consider:

```
requires (T a, T b)  {
  // ...
};
```

Like the axiom in Common, arguments to the requires clause allow us to introduce variables needed to express those requirements. In this way, the requirements are quantified over all

values a and b of type T. For example, the following requirement states that we can compare two T values using ==, and that the result can be used to initialize a bool:

```
bool { a == b };
```

The curly braces are the C++11 uniform initialization syntax. We say that the requirement is "checkable" because a C++11 compiler can perform template argument substitution and determine if the expression results in a substitution failure or not. The check can be implemented as a type trait.

We require "convertible to bool" for the result of == rather than exactly bool to cater for conversions of all sorts. One reason for this degree of freedom is that many legacy libraries include classes whose relational operators return int. Examples in production code can be easily found. One such example is an early version of the QChar class (1.5 and earlier, at least) (Nokia Corporation, 2011).

```
class QChar
{
  friend int operator==(QChar c1, QChar c2);
  friend int operator!=(QChar c1, QChar c2);
};
```

We should be able to use this class in our standard algorithms, despite the fact that the operator does not return a bool. Incidentally, this class ships with the current release of Doxygen (van Heesch, 2011), so it is not just an example of forgotten and unused legacy code.

The notation for syntactic requirements is similar, but not identical, to the use-pattern approach discussed in Dos Reis and Stroustrup (2006). That paper demonstrates that such a notation can be mathematically sound. We use it because we feel that it most directly expresses requirements and axioms. However, the meaning of the requirements is our primary concern. If there is a better (terser, more comprehensible) notation, we can use that instead.

We have divided the semantics of the EqualityComparable concept into three axiom blocks. The states the meaning of ==. It says that the expression a == b is true if and only if the expression eq(a, b). In other words, the == operator must implement the specification of equality for the type T.

Axioms in concepts often state *equations* between two expressions. Here, the <=> operator equates the expression on the left with that on the right. Because the expressions are logically equivalent, one can be substituted for the other (although eq has no meaning outside of axioms because it can't be evaluated). This directly supports equational reasoning.

The second axiom specification states the relational semantics of the == operator; it must behave as an equivalence relation:

```
axiom {
  a == a;
  a == b => b == a;
  a == b && b == c => a == c;
}
```

These are the reflexive, symmetric, and transitive properties, respectively. The => operator is the *implication* operator. If the left-hand side is true, the right-hand side must also be true, or if the left-hand side is false, then the entire expression is vacuously true.

The purpose of axioms is to specify the semantic properties of expressions required by a concept. They describe the behavior of those expressions when evaluated with well formed arguments. We assume that any variables introduced in a requires or axiom declaration are well formed.

Obviously, not all arguments will be well formed for any given type. For example NaN is not well-formed data and does not satisfy any of the axioms of the equivalence relation above. Similarly, the expression *p is not well formed when p == nullptr. Because it results in undefined

behavior, it would be inappropriate to consider that value when describing the general semantics of equality. Note that these exceptional cases do not contradict the axioms of the concept. This is not proof, for example, that IEEE 754 floating point types are not EqualityComparable. The fact that *some* representations of the format can be interpreted as non-values, does not imply any properties of valid interpretations.

Programmers should be aware that any program that casually relies on the results of operations on ill-formed data (such as NaN) inherently invoke undefined behavior. Preconditions are a traditional method for guarding against undefined behavior. Exceptions can also be thrown to terminate an operation that cannot compute a result that satisfies its postconditions. For example, an exception might be thrown if there is insufficient memory to concatenate two strings (e.g. bad_alloc). As with NaN, this is not proof that string concatenation is not an associative operation.

The last axiom of the EqualityComparable concept is:

```
axiom { a != b <=> !(a == b); }
```

It defines the meaning != in terms of ==: a != b if and only !(a == b). Because the two are equal, we are free to substitute one for the other when reasoning about a program. Note that the interpretation of the operator ! on the right-hand side is the built-in operator for bool types. The logical basis of axiomatic specifications is the built-in Boolean algebra.

So, EqualityComparable<T> is true if T

1. has == and != with result types that can be converted to bool
2. == compares for true equality
3. == is reflexive, symmetric, and transitive
4. != is the complement of ==

However, the compiler can only check the first of these conditions. The rest must be verified through other means (i.e. manually).

Concepts, like functions, can be overloaded. We use this feature to build mathematically meaningful descriptions of "cross-type" equality and order.

```
concept EqualityComparable<EqualityComparable T1, EqualityComparable T2> =
  Common<T1, T2> &&
  EqualityComparable<CommonType<T1, T2>> &&
  requires(T1 a, T2 b) {
    bool { a == b };
    bool { b == a };
    bool { a != b };
    bool { b != a };
    axiom {
      using C = CommonType<T1, T2>;
      a == b <=> C{a} == C{b};
      a != b <=> C{a} != C{b};
      b == a <=> C{b} == C{a};
      b != a <=> C{b} != C{a};
    }
  };
```

We use && to combine requirements; A && B is true if both A and B are true.

This concept extends the notion of equality comparison for a single type to equality comparisons involving different different types. In order for two different types to be equality comparable, the following conditions must hold:

- EqualityComparable<T1>

- EqualityComparable<T2>
- Common<T1, T2>
- EqualityComparable<CommonType<T1, T2>>
- the requires clause and associated semantics

The semantics of equality comparison are defined in terms of equivalences. Any combination of comparisons involving objects of types T1 and T2 are equivalent to comparisons involving the common type.

Recall that the Common concept requires conversions to the common type to preserve the original values. Without this guarantee, we would not be able to assert the correspondence between the heterogeneously typed operations with their homogeneously typed equivalents.

It is important to note that, while convertibility is a principal requirement of CommonTypes, conversions are not *required* in contexts where related types are compared. Consider the following function:

```
template<typename T, typename U>
requires EqualityComparable<T, U>
void f(const T& a, const U& b)
{
  assert(a == b);
}
```

During instantiation, the resolution of == can entail two possible outcomes:

- If there is an overload of == for T and U, then that will be chosen.
- If there is no overload for T or U, but a conversion sequence exists that allows an overload to be selected, then the conversion sequence will be applied.

In actuality, the lookup is resolved before the expression is analyzed. The same overload used in the function body must be the same as that resolved when the EqualityComparable concept is checked, including any conversion sequences, if needed. However, no conversion overhead is required if the appropriate overloads are defined. That is, comparing a string and const char* does not invoke a conversion.

The value-oriented style of the STL is predicated on the notion of *regularity*: that objects behave in the expected and consistent ways, and that they behave similarly to built-in value types like int or float. These kinds of types support familiar forms of object construction and destruction; they can be default initialized, moved, and copied, and dynamically allocated and deleted.

Intuitively, the Semiregular concept describes types that behave in regular ways except that they might not be comparable using ==. Examples of Semiregular types include: all built-in C++ integral and floating point types, all trivial classess (trivial in the sense of the C++ standard (C++ Standard, class)), std::string, and standard containers of copyable types (e.g., vector<string>). The Semiregular concept is defined as:

```
concept Semiregular<typename T> =
  requires object (T a) {
    // Address-of
    T* == {&a};
    axiom { &a == addressof(a); }

    // Non-volatility
    axiom { eq(a, a); }
  } &&
```

```
requires destructible (T a) {
  // Destruction
  a.~T();
  noexcept(a.~T());
} &&
requires initialization (T a, T b, T c) {
  T{};          // Default construction
  T{a};         // Copy construction
  T& == {a = b}; // Copy assignment

  axiom copy_semantics {
    eq(T{a}, a);
    eq(a = b, b);
  }

  axiom move_semantics {
    eq(a, b) => eq(T{move(a)}, b);
    eq(b, c) => eq(a = move(b), c);
  }
} &&
requires allocation (size_t n) {
  T* == { new T };
  delete new T;

  T* == { new T[n] };
  delete[] new T[n];
};
```

The definition of this concept is similar to those above, except that we have named some of the requirements:

```
requires object (T a) { ... }
requires destructible (T a) { ... }
requires initialization (T a, T b, T c)
requires allocatable () { ... }

axiom copy_semantics { ... }
axiom move_semantics { ... }
```

It is often useful, especially for larger concepts, to break their syntactic and semantic requirements into smaller, more readily digested components. The names themselves have no actual meaning within the program. They are simply labels that can be used to document sets of related requirements. We could equivalently leave out the requires keyword and just write:

```
object (T a) { ... } &&
destructible (T a) { ... } &&
initialization (T a, T b, T c) &&
allocatable () { ... };
```

We know that the requirements in those blocks are syntactic and must be validated because they are not prefixed with the axiom keyword. We could also have just written requires without names, just like we did with the EqualityComparable concepts. Note, however, that you cannot omit the axiom identifier for the copy_semantics and move_semantics axioms.

The "object" requirements of the Semiregular types are that:

- on object of this type can have its address can be taken, and
- the result of &a is a pointer to T, and

- the meaning of &a is equivalent to addressof(a).

Overloading the address-of operator to do something other than return this makes a type irregular. Overloading the comma operator also makes a type irregular, but we don't have a convenient notation for expressing that particular prohibition.

The the non-volatility axiom in this set of requirements ensures that volatile types are not considered to be Semiregular. Objects whose states are modified by external processes do not describe regular types and cannot be reasoned about using conventional logics.

The destructible requirements state objects of type T must be destructible (stated using a pseudo-destructor call (C++ Standard, expr.pseudo)) and that the destructor must not propagate exceptions. The second requirement is enforced using the noexcept operator (C++ Standard, expr.unary.noexcept). In a requires block, the noexcept operator acts like a concept. If the computed result of the operator is false, then the requirement is not satisfied.

The initialization requirements describe how objects of type T can be construction and assigned. The syntax T == {&a} constrains the result type of an expression. It is related to the usual initializer syntax used to declare variables (C++ Standard, decl.init). We can express constraints on result types in three ways:

```
T{expr}     // expr is explicitly convertible to T
T = {expr}  // expr is implicitly convertible to T
T == {expr} // expr has the same type as T
```

The explicit and implicit conversion requirements re-use the initialization syntax. For example, a variable declaration T x{expr} declares a variable x whose value is explicitly converted from expr. Like explicit conversions, a declaration T x = {expr} will implicitly convert the result of expr into T. The == initialization is equivalent to the following requirements:

```
T{expr};
Same<decltype(expr), T>;
```

We think T == {expr} is a more uniform approach to writing the requirement than enumerating Same constraints.

The axioms of copy semantics are straightforward: copies compare equal to their originals. The move semantic axioms guarantee state the semantics of move construction and assignment. The constructed or assigned object is equal to the value of the original prior to the move operation. The moved-from object is partially formed after the move construction or assignment.

Note that copy semantics do not preclude shallow copies. If they did, we might not be able to conclude that pointers are copyable. Shallow-copied types may be regular with respect to copy and move semantics, but they probably have some operations that are not equality preserving. This is definitely the case with some InputIterators such as istream_iterator.

The equations in the copy- and move- semantics blocks are written using the eq predicate because the Semiregular concept does not require T to be EqualityComparable. A concept cannot use syntax that it has not required. In general, we prefer to write semantics in terms of == when possible and only use eq when the operation is not available. For example, the comparison of pointers (&a == addressof(a)) is perfectly fine because the syntax and semantics of pointers are established by the C++ programming language; we know for a fact that they are equality comparable (i.e., for all types T, EqualityComparable<T*> is true).

Semiregular types can be used to declare variables and temporaries. The Allocatable requirements also give us the ability to dynamically allocate (and delete) objects and temporary buffers. The requirement is parameterized over a size_t value (n), which is used as an argument to the array allocation syntax. Note that this does not guarantee that allocation will succeed for extremely large values of n, only that it is syntactically possible. The actual limits of dynamic allocation are determined by the host system, as described by the standard (C++ Standard, expr.new/7).

These basic facilities are used in many, many algorithms and are so fundamental that we include them as a single concept. Unfortunately, this over-constrains a number of algorithms by requiring default construction and copy semantics when the algorithm does not rely on them. We describe an alternative design that avoids these issues in Appendix D.

The Semiregular concept demonstrates an important distinction between the notions of value type and object type. Here, we clearly expect the template argument T to represent a value type: non-reference and unqualified. Constants (const T) and constexpr objects do not have Semiregular types since they cannot be assigned to. Volatile types (volatile T) are not Semiregular since their objects' states may by changed externally. Reference types types are not Semiregular because the syntax used for copy construction does not actually create a copy; it binds a reference to an object. Proper use of the Semiregular concept requires that it is evaluated over value types, not object types. This is true of every concept from this point forward in the report.

The Regular concept unifies the Semiregular and EqualityComparable concepts. Regularity is defined as:

```
concept Regular<Semiregular T> = EqualityComparable<T>;
```

We could also have chosen to write it this way:

```
concept Regular<typename T> = Semiregular<T> && EqualityComparable<T>;
```

The two definitions are equivalent. By convention, we write the "strongest" requirement as the template argument type, so we tend to prefer the first definition in this report.

Note that because EqualityComparable is required, we could conceivably rewrite the axioms of the Semiregular concept for Regular using == instead of eq. For example, the axioms of copy semantics become:

```
T{a} == a;
(a = b) == b;
```

The meaning is, of course, the same since EqualityComparable requires == and eq to be logically equivalent.

A TotallyOrdered type is an EqualityComparable type equipped with inequality operators <, >, <=, and >= and whose values are totally ordered.

```
concept TotallyOrdered<EqualityComparable T> =
  requires (T a, T b, T c) {
    bool { a < b };
    bool { a > b };
    bool { a <= b };
    bool { a >= b };
    axiom {
      !(a < a);
      a < b => !(b < a);
      a < b && b < c => a < c;
      a < b || b < a || a == b;
    }
    axiom {
      a > b <=> b < a;
      a <= b <=> !(b < a);
      a >= b <=> !(b > a);
    }
  };
```

As with EqualityComparable, the TotallyOrdered concept requires a number of relational operators whose results are convertible to bool. The semantics of TotallyOrdered are written in two axiom blocks. Within the body of a requires clause, axiom blocks can be written as consecutive

compound statements. We could write the axioms defining the properties of < in a more stylized way by naming them.

```
requires (T a, T b, T c) {
  // ...
  axiom irreflexive { !(a < a); }
  axiom asymmetric { a < b => !(b < a); }
  axiom transitive { a < b && b < c => a < c; }
  axiom total { a < b || b < a || a == b; }
}
```

This has a nice property of stating exactly what properties are being described. The second axiom block connects the meaning of < to the other inequality operators >, <=, and >=.

A general requirement for total ordering seems like it might be overly strict. Why not allow the operator < to define partially ordered types? Our aim is to align the specifications with programmer's intuitions about the meaning of operations. The connection between the relational operator < and total orders is taught in grade school. We think it is important to retain, if not emphasize, these expectations (while still being mathematically precise, of course).

Just like the EqualityComparable concept, we can overload TotallyOrdered to support a mathematically sound extension for different types.

```
concept TotallyOrdered<TotallyOrdered T1, TotallyOrdered T2> =
  TotallyOrdered<CommonType<T1, T2>> &&
  EqualityComparable<T1, T2> &&
  requires (T1 a, T2 b) {
    bool { a < b };
    bool { a > b };
    bool { a <= b };
    bool { a >= b };
    bool { b < a };
    bool { b > a };
    bool { b <= a };
    bool { b >= a };
    axiom {
      using C = Common<T1, T2>;
      a < b <=> C{a} < C{b};
      a > b <=> C{a} > C{b};
      a <= b <=> C{a} <= C{b};
      a >= b <=> C{a} >= C{b};
      b < a <=> C{b} < C{a};
      b > a <=> C{b} > C{a};
      b <= a <=> C{b} <= C{a};
      b >= a <=> C{b} >= C{a};
    }
  };
```

Two types are TotallyOrdered if they are

- both TotallyOrdered and EqualityComparable,
- they share an EqualityComparable, TotallyOrdered common type, and
- any inequality involving arguments of type T1 or T2 is equal to the same expression involving the common type.

For example, TotallyOrdered<int, long> is true because the language gives rules for comparing them in terms of their common type (which happens to be long). TotallyOrdered<long, double> is true for the same reason.

Unfortunately, the inheritance of C typing rules implies that signed int and unsigned int are TotallyOrdered, despite the fact that comparing signed and unsigned values is a known source of bugs. Fortunately, C++ compilers will warn you when comparing these values, assuming that the appropriate warnings are enabled.

3.4 Function Concepts

Function concepts describe requirements on function types. There are eight in total (six and two overloads):

```
concept Function<typename F, typename... Args>
concept RegularFunction<typename F, typename... Args>

concept Predicate<typename P, typename... Args>
concept Relation<typename R, typename T>
concept Relation<typename R, typename T, typename U>

concept UnaryOperation<typename Op, typename T>
concept BinaryOperation<typename Op, typename T>
concept BinaryOperation<typename Op, typename T, typename U>
```

A Function is a type whose objects can be called over a (possibly empty) sequence of arguments. Functions are not Semiregular types; they may not exhibit the full range of capabilities as built-in value types. Minimally, we can expect to copy- and move-construct Function types, but not default construct or copy- and move-assign them. The Function concept is defined as:

```
concept Function<typename F, typename... Args> =
  requires object (T a) {
    T* == {&a};
    axiom { &a == addressof(a); }
    axiom { eq(a, a); }
  } &&
  requires destructible (T a) {
    a.~T();
    noexcept(a.~T());
  } &&
  requires initialization (T a, T b, T c) {
    T{a};
    axiom {
      eq(T{a}, a);
      eq(a, b) => eq(T{move(a)}, b);
    }
  } &&
  requires allocation () {
    T* == { new T{a} };
    delete new T{a};
  } &&
  requires callable(F f, Args args...) {
    ResultType<F, Args...> == { f(args...) };
    axiom {
      not_equality_preserving(f(args...));
    }
  };
```

This is a *variadic concept*; the template parameter pack Args denotes a sequence of argument types: the domain of the function. We read this as F must be callable with the argument types, Args.... The concept places no constraints on the argument types.

The object requirements are the same as those in the Semiregular concept, but the initialization requirements have changed. Function objects are not required to be copy (or move) assignable. In fact, lambda closure types explicitly deletes the default constructor and copy assignment operator (C++ Standard, expr.lambda/19). Because Functions are not default constructible or assignable, they are not Semiregular.

We could have refactored the concept design to define individual concepts for features like Destructible, CopyConstructible, and MoveConstructible. However, this moves away from our design ideals: concepts that represent abstractions in the application domain. These concepts describe individual features, not abstractions. It is possible that these concepts may be required by future designs, but we did not need them to express the requirements of the STL algorithms.

The syntactic requirements of Functions can be expressed in a single statement:

```
requires(F f, Args args...) {
  ResultType<F, Args...> == { f(args...) };
}
```

The requirement is quantified over a function, function object, or lambda expression f having type F, and a function parameter pack args whose types are given by the type sequence Args. There are three syntactic requirements in this clause.

- The type function ResultType<F, Args...> must be defined for all Function types that are callable with argument types Args....
- The expression f(args...) is valid; we can call f with the arguments given in the args parameter pack.
- The call f(args...) has the Same type as ResultType.

Taken as a whole, the requirement says that f can be called with the specified arguments and the type resulting from the invocation is called F's ResultType. Note that a void result type is perfectly acceptable for Function types.

Semantically, a Function is not required to preserve equality. This is clearly stated by the not_equality_preserving predicate. Recall that we assume required expressions are equality preserving by default. Here, we have to explicitly indicate that the call expression for Function types does not require this property.

A RegularFunction is a Function that is equality preserving:

```
concept RegularFunction<typename F, typename... Args> =
  Function<F, Args ...> &&
  axiom (F f, Args... args) {
    equality_preserving(f(args...));
  };
```

Regular functions—those having predictable and reasonable side effects—provide the semantic foundation for the specification of Predicates, Relations (§3.4.1), and operations (§3.4.2).

3.4.1 Predicates

A Predicate is a RegularFunction whose result type is convertible to bool.

```
concept Predicate<typename P, typename... Args> =
  RegularFunction<P, Args...> &&
  Convertible<ResultType<P, Args...>, bool>;
```

Because P is required to be a Function (by way of RegularFunction), we are guaranteed that ResultType<P, Args...> is defined.

Despite the fact that RegularFunctions have regular semantics, we do not constrain the argument types in its specification. Trying to do so may have unintended consequences. It makes it

impossible, for example, to call any algorithm taking a Predicate on a sequence of non-Semiregular types such as the resource-like unique_ptr. It seems unreasonable to disallow programs like this:

```
vector<unique_ptr<employee>> v = { ... };
auto i = find(v.begin(), v.end(),
              [](const unique_ptr<employee>& p) {
                return !p || p->name() == "";
              });
```

Care must be taken when specifying requirements not to accidentally preclude a set of otherwise valid template arguments. We have found that constraining the argument types of RegularFunctions is an easy way to achieve that undesirable goal.

A Relation is a binary Predicate.

```
concept Relation<typename P, typename T> = Predicate<P, T, T>;
```

The STL is concerned with two kinds of relations: *equivalence relations* which generalize equality, and *strict weak orderings* which generalize total orderings. Although the C++0x proposals included concepts describing these properties, we do not. In C++, these properties can only be defined for particular objects, not types. This is because there are many functions with type bool(T, T) that are neither equivalence relations nor strict weak orderings. The precise semantics of these kinds of Relations is defined in §B.1.2.

As with the EqualityComparable and TotallyOrdered concepts, we can extend its definition to different types:

```
concept Relation<typename R, typename T1, typename T2> =
  Relation<R, T1> &&
  Relation<R, T2> &&
  Common<T1, T2> &&
  Relation<R, CommonType<T1, T2>> &&
  requires(R r, T1 a, T2 b) {
    bool { r(a, b) };
    bool { r(b, a) };
    axiom {
      using C = CommonType<T1, T2>;
      r(a, b) <=> r(C{a}, C{b});
      r(b, a) <=> r(C{b}, C{a});
    }
  };
```

A Relation can be defined on different types if

- T1 and T2 share a CommonType C,
- the Relation R is defined for all combinations of those types, and
- any invocation of r on any combinations of types T1, T2, and C is equivalent to an invocation r(C{a}, C{b}).

These are strong requirements, but we feel that they are justified. The additional requirements make it possible to reason about the semantics of cross-type relations; they make it possible to describe such relations as equivalence relations or strict weak orderings.

3.4.2 Operations

We provide three concepts for describing function concepts related to numeric operations. An operation is a RegularFunction with a homogeneous domain whose result type is convertible to its domain type.

```
concept UnaryOperation<typename Op, typename T> =
  RegularFunction<Op, T> &&
  Convertible<ResultType<Op, T>, T>;

concept BinaryOperation<typename Op, typename T> =
  RegularFunction<Op, T, T> &&
  Convertible<ResultType<Op, T, T>, T>;

concept BinaryOperation<typename Op, typename T1, typename T2> =
   BinaryOperation<Op, T1> &&
   BinaryOperation<Op, T2> &&
   Common<T1, T2> &&
   BinaryOperation<Op, CommonType<T1, T2>> &&
   requires(Op op, T1 a, T2 b) {
     using C = CommonType<T1, T2>;
     C = {op(a, b)};
     C = {op(b, a)};
     axiom {
       eq(op(a, b), op(C{a}, C{b}));
       eq(op(b, a), op(C{b}, C{a}));
     }
   };
```

The UnaryOperation concept defines requirements for operations on a single value; it is a RegularFunction that takes a single argument of type T and whose result type is convertible to T. The BinaryOperation defines requirements for operations on two arguments. The actual requirements are similar to UnaryOperation.

We also extend BinaryOperation to the case where the argument types differ. This is done by requiring the operation to be defined on the common type and that invocations of op over all combinations of argument types T1 and T2 are equivalent to invocations on the common type.

These concepts are not used directly by the STL algorithms in the STL. We include them because they will (in the future) be useful when defining requirements for the standard numeric algorithms (C++ Standard, numeric.ops).

3.5 Iterator Concepts

Iterators are one of the fundamental abstractions of the STL. They generalize the notion of pointers. There are 12 concepts related to iterators. The first five concepts are properties of iterators, while the last 7 describe iterator abstractions.

```
// Iterator properties:
concept Readable<Semiregular I>
concept MoveWritable<typename T, Semiregular Out>
concept Writable<typename T, Semiregular Out>
concept IndirectlyMovable<Readable I, Semiregular Out>
concept IndirectlyCopyable<Readable I, Semiregular Out>

// Incrementable types:
concept WeaklyIncrementable<Semiregular I>
concept Incrementable<Semiregular I>

// Iterator abstractions:
concept WeakInputIterator<Incrementable I>
concept InputIterator<WeakInputIterator I>
concept ForwardIterator<InputIterator I>
concept BidirectionalIterator<ForwardIterator I>
concept RandomAccessIterator<BidirectionalIterator I>
```

This iterator design is different than that found in the C++ standard. It has (many) more concepts and does not include an explicit notion of output iterators. In part, some of the added complexity comes from the fact that we differentiate between iterators that require equality comparison (those in bounded ranges) and those that do not (those in weak ranges). The addition of move semantics also introduces a new concept (MoveWritable).

3.5.1 Iterator Properties

Iterator properties deal with reading values from and writing values to iterators. The Readable concept defines the basic properties of *input iterators*; it states what it means for a type to be readable.

```
concept Readable<Semiregular I> =
  requires(I i) {
    ValueType<I>;
    const ValueType<I>& = { *i };
  };
```

A Readable type has an associated value type, which can be accessed using ValueType<T>. The actual property that makes the type Readable is the fact that it can be dereferenced, and we can refer to that value using a const reference.

The reason that we require convertibility to a reference type instead of to the ValueType is that the latter can imply a copy requirement on the destination type. Suppose we have the following:

```
vector<unique_ptr<T>> v = { ... };
auto i = v.begin();
unique_ptr<T> p = { *i }; // error: not copyable
```

This results in a compiler error because unique_ptrs are not copyable. Unfortunately, this is the exact requirement that we would have stated for Readable if we required convertibility to the ValueType instead of a reference.

This has a serious implication on the way that we write algorithms and their requirements. If an algorithm uses a temporary buffer to store intermediate results, then it must explicitly require that the iterator's ValueType is Semiregular. An example of this is the relaxed version of is_sorted_until described in §2.3.1.

Unlike the C++ standard's InputIterator concept, the Readable concept does not require the -> operator for iterators. Using the -> syntax requires in an algorithm requires that you know

1. that the iterator's value type is a class or union type, and
2. the name of the member on the right hand side.

Although, we could test for the first property within the concept, there is no way to evaluate the second without additional information. Furthermore, we know of no uses of the -> operator in any STL implementations. As such, we omit the requirement from the concept.

Writable and MoveWritable describe the fundamental properties of *output iterators*. The MoveWritable concept describes a requirement for moving a value into an iterator's referenced object.

```
concept MoveWritable<typename T, Semiregular Out> =
  requires (T value, Out o) {
    *o = move(value);
    axiom (T other) {
      Readable<Out> && Same<ValueType<Out>, T> =>
        is_valid(*o = move(value)) => eq(value, other) =>
          (*o = move(value), eq(*o, other)));
```

```
    }
};
```

The concept describes a *relation* between an iterator and the type whose value is being moved. This is, in some respects, similar to the OutputIterator described in the C++0x proposals (C++ Standards Committee, 2009), except that it does not include any requirements for increment.

The axiom, like the requires clause is parameterized. This is convenient notation for quantifying an axiom over variables that have not been previously introduced. Within the axiom, the is_valid predicate is a compiler intrinsic that returns true if the expression is defined for its given operands. This guarantees that writing *o will not result in undefined behavior; it won't accidentally power down your workstation or melt your CPU. is_valid is a kind of universal precondition that lets us exclude unnamed values from the definition space of the expression. Dereferencing an iterator is only valid if an iterator is in a range but not equal to its limit. We use this predicate extensively when describing the semantics of iterators. Note that real validity can be derived from the preconditions of an algorithm.

The meaning of the required expression are defined conditionally; we can only state the meaning if the Out iterator is Readable and ValueType<Out> and T are the same type. The axiom is (unsurprisingly) similar to that describing move assignment in the Semiregular concept (§3.3). After evaluating *o = move(value), the value referred to by *o is equal to other. The moved-from object value is left partially formed. Note that the result of the entire expression is unconstrained.

The Writable concept describes a requirement for writing a value to a dereferenced iterator. It is defined as:

```
concept Writable<typename T, Semiregular Out> =
  MoveWritable<T, Out> &&
  requires(T value, Out o) {
    *o = value;
    axiom {
      Readable<Out> && Same<ValueType<Out>, T> =>
        (is_valid(*o = value) => (*o = value, eq(*o, value)));
    }
};
```

The Writable concept extends the requirements of MoveWritable to also include copy assignment through an iterator. Recall that copy and move semantics are related. If you can copy value through *o, then you can also move it. The semantics of Writable are analogous to those of copy assignment.

Unlike Readable, the MoveWritable and Writable concepts do not have a built-in notion of ValueType. It is perfectly reasonable for an algorithm to have multiple Writable requirements taking different value types. The Mergeable concept (§3.6) and its related algorithms (§2.3.3 and §2.3.4) do exactly this.

We introduce several "indirect" concepts to describe relationship between iterator types. These are:

```
concept IndirectMovable<Readable I, Semiregular Out> =
  MoveWritable<ValueType<I>, Out>;

concept IndirectlyCopyable<Readable I, Semiregular Out> =
  Writable<ValueType<I>, Out>;
```

The IndirectlyMovable and IndirectlyCopyable concepts describe copy and move relationships between the values of an input iterator, I, and an output iterator Out. For an output iterator out and an input iterator in, their syntactic requirements expand to:

- IndirectlyMovable requires *out = move(*in)

- IndirectlyCopyable requires *out = *in

3.5.2 Incrementable Types

The concepts in this section describe incrementable types. All iterators have, as a basic trait, the property that they can be incremented. A weakly incrementable type, represented by the WeaklyIncrementable concept, represents a kind of type that can be pre- and post-incremented. It describes the behavior of pre- and post-increment for both input and output iterators that are not also forward iterators. WeaklyIncrementable types are used in weak input and output ranges where equality comparison and equality preservation are not required.

```
concept WeaklyIncrementable<Semiregular I> =
  requires {
    // Associated types:
    DistanceType<I>;
    Integral<DistanceType<I>>;
  } &&
  requires (I i) {
    // Pre-increment:
    I& == {++i};
    axiom {
      // if valid, ++i moves i to the next element
      not_equality_preserving(++i);
      is_valid(++i) => &++i == &i;
    }

    // Post-increment:
    i++;
    axiom {
      // if valid, i++ moves i to the next element
      not_equality_preserving(i++);
      is_valid(++i) <=> is_valid(i++);
    }
  };
```

Here, we divide the requirements into two groups. The first grouping specifies an associated type: DistanceType. This is required to be an Integral type; no judgment is made as to whether this should be signed or unsigned in this context. Note that we could have omitted the statement DistanceType<I>. This statement:

```
Integral<DistanceType<I>>;
```

is sufficient to induce a requirement on the definition of DistanceType<I>..

The DistanceType is a numeric type that can represent the largest possible number of applications of the increment operator for the iterator type I. This is the same as the "difference type" in the C++11 design. We prefer "distance" over "difference" since the type is more generally associated with the distance operation, not subtraction.

WeaklyIncrementable types allow both pre- and post-increment, and both operations move the operand to the next element. However, neither increment operation is required to preserve equality, and the result type of the post-increment operation is unspecified. This makes describing the relationship between the two operators somewhat difficult. At best, we can say two things:

1. they have equivalent effects because they have the same documented behavior, and
2. if one operation is valid, so is the other.

It is debatable whether WeaklyIncrementable should actually require a post-increment operation. The semantics of the post-increment operator are vague, and the operator is not frequently used in STL implementations.

The Incrementable concept matches our more common understanding of incrementable types. They are Regular types where both pre- and post-increment are equality preserving operations, and the syntax and semantics of the post-increment operation are given stronger meanings. The regularity of pre- and post-increment allows multiple passes over a range of Incrementable objects. This is traditionally referred to as the "multi-pass" property of iterators. The concept's definition is:

```
concept Incrementable<Regular I> =
  WeaklyIncrementable<I> &&
  requires (I i) {
    // Pre-increment:
    axiom {
      equality_preserving(++i);
    }

    // Post-increment:
    I == {i++};
    axiom (I j) {
      equality_preserving(i++);
      is_valid(i++) => (i == j => i++ == j);
      is_valid(i++) => (i == j => (i++, i) == ++j);
    }
  };
```

Whereas the result of post-incrementing a WeaklyIncrementable type is unspecified, the Incrementable concept requires it to be exactly I. The axioms state the equality-preserving nature of the pre- and post-increment operators and properties of the result of post-increment. The result of post-increment is the previous iterator, and the effect of post-incrementing an iterator is the same as pre-incrementing it.

3.5.3 Iterator Types

Iterators are incrementable and readable types. For example, a WeakInputIterator is a WeaklyIncrementable and Readable type. WeakInputIterators are used in weak ranges so they are not required to be EqualityComparable; the concept's definition follows:

```
concept WeakInputIterator<WeaklyIncrementable I> =
  Readable<I> &&
  requires (I i) {
    IteratorCategory<I>;
    Derived<IteratorCategory<I>, weak_input_iterator_tag>;
    Readable<decltype(i++)>;
  };
```

The concept adds two additional static requirements: the type function IteratorCategory must be defined for I, and the aliased type must be derived from the tag class weak_input_iterator_tag. The iterator category is an artifact of the original STL iterator design. Historically, iterator categories were used to differentiate different specializations based on iterator kind. While C++11 makes it possible to evaluate all static requirements (using type trait hackery), we still need to differentiate some concepts based on their semantic requirements. The iterator category solves that problem for us.

The requirement Readable<decltype(i++)> guarantees that we can dereference the result of the post-increment operator, but it does not say what that type actually is. Some iterators have

post-increment operators that return *proxies*: objects used to provide access to the previous state but are not actually iterators.

An InputIterator is an EqualityComparable WeakInputIterator. Every readable bounded range requires, at least, an InputIterator.

```
concept InputIterator<WeakInputIterator I> =
  EqualityComparable<I> &&
  Derived<IteratorCategory<I>, input_iterator_tag>;
```

The definition is straightforward. Like the WeakInputIterator, it includes a derivation requirement for its iterator category. While the concept indirectly requires WeaklyIncrementable and EqualityComparable, it does not require that increment is an equality preserving operation. This means that a range of InputIterators can only be traversed once.

A ForwardIterator is an Incrementable InputIterator. Because its increment operation is equality-preserving, ForwardIterators permit multiple passes over the data being traversed. ForwardIterator types abstract the notion of traversal of singly-linked lists. They also tend to refer to data that persists beyond the lifetime of the iterator's current position. Incrementing a ForwardIterator does not have any side effects that impact the regularity of its operations. Its definition is:

```
concept ForwardIterator<InputIterator I> =
  Incrementable<I> &&
  Derived<IteratorCategory<I>, forward_iterator_tag>;
```

A BidirectionalIterator is a ForwardIterator that supports the decrement operation; BidirectionalIterators abstract the traversal of doubly-linked lists. The defining concept includes the two decrement operators, which are defined analogously to the increment operators.

```
concept BidirectionalIterator<ForwardIterator I> =
  Derived<IteratorCategory<I>, bidirectional_iterator_tag> &&
  requires decrement (I i, I j) {
    // Pre-decrement:
    I& == { --i };
    axiom { is_valid(--i) => &--i == &i; }

    // Post-decrement:
    I == { i-- };
    axiom {
      is_valid(--i) <=> is_valid(i--);
      is_valid(i--) => (i == j => i-- == j);
      is_valid(i--) => (i == j => (i--, i) == --j);
    }
  } &&
  axiom increment_decrement (I i, I j) {
    is_valid(++i) => (is_valid(--(++i)) && (i == j => --(++i) == j));
    is_valid(--i) => (is_valid(++(--i)) && (i == j => ++(--i) == j));
  };
```

In this concept, we have split its definition into two parts: a named Decrement requirement that states the syntax and semantics of pre- and post-decrement, and a named axiom (Increment_decrement) that describes the relationship between the increment and decrement operators. Axioms can be named in the same way that requirements are named, except that the identifier axiom must still be used to introduce the clause.

The syntactic and semantic requirements in the Decrement clause can be paired, one-to-one with the increment axioms in ForwardIterator. Note that we don't have to explicitly state the equality-preserving nature of the operator because we assume that all required operations are equality preserving unless otherwise stated.

The Increment_decrement axiom clause relates the increment and decrement operation through a pair of mutual equations. These equations say that if we reach an iterator by incrementing then we can reach its predecessor by decrementing and vice versa.

A RandomAccessIterator is a TotallyOrdered BidirectionalIterator that can advance some number of steps, in either direction, in constant time, leading to a number of new syntaxes. The distance between RandomAccessIterators can also be computed in constant time by subtracting two values. The concept, while extensive, has straightforward requirements:

```
concept RandomAccessIterator<BidirectionalIterator I> =
  TotallyOrdered<I> &&
  Derived<IteratorCategory<I>, random_access_iterator_tag> &&
  SignedIntegral<DistanceType<I>> &&
  difference (I i, I j) {
    DifferenceType<I> == { i - j };
    SignedIntegral<DifferenceType>;
    Convertible<DistanceType, DifferenceType>;
    axiom {
      is_valid(distance(i, j)) <=> is_valid(i - j) && is_valid(j - i);
      is_valid(i - j) => (i - j) >= 0 => i - j == distance(i, j);
      is_valid(i - j) => (i - j) <  0 => i - j == -distance(i, j);
    }
  } &&
  advance (I i, I j, DifferenceType<I> n) {
    // Addition:
    I& == { i += n };
    I == { i + n };
    I == { n + i };
    axiom {
      is_valid(advance(i, n) <=> is_valid(i += n);
      is_valid(i += n) => i += n == (advance(i, n), i);
      is_valid(i += n) => &(i += n) == &i;
      is_valid(i += n) => i + n == (i += n);

      // Commutativity of pointer addition
      is_valid(i + n) => i + n == n + i;

      // Associativity of pointer addition
      is_valid(i + (n + n)) => i + (n + n) == (i + n) + n;

      // Peano-like pointer addition:
      i + 0 == i;
      is_valid(i + n) => i + n == ++(i + (n - 1));
      is_valid(++i) => (i == j => ++i != j);
    }

    // Subtraction:
    I& == { i -= n };
    I ==  { i - n };
    axiom {
      is_valid(i += -n) <=> is_valid(i -= n);
      is_valid(i -= n) => (i -= n) == (i += -n);
      is_valid(i -= n) => &(i -= n) == &i;
      is_valid(i -= n) => (i - n) == (i -= n);
    }
  } &&
  subscript (I i, DifferenceType<I> n) {
```

```
    ValueType<I> = { i[n] };
    axiom {
      is_valid(i + n) && is_valid(*(i + n)) => i[n] == *(i + n);
    }
  };
```

The difference requirements vary from what is required by the C++ standard. There, an iterator's difference type describes both the result of the distance algorithm and arithmetic subtraction of random access iterators. The distance algorithm requires a bounded range, [first, last), which should imply that its result is non-negative (last must be reachable from first by a series of increment operations). The standard mandates a different definition for random access iterators: distance(i, j) == j - i. We see this as a specification error; the guarantees of the distance operation have been weakened for an iterator specialization.

In our design, we consider the two operations to be distinct. The result type of iterator subtraction is called the DifferenceType. It must encode the largest possible DistanceType *and* their additive inverses (hence the SignedIntegral requirement). In other words, if n is the distance between i and j, then the difference type must be able to represent both n and $-n$. This is supported by the requirement that the DistanceType be convertible to the DifferenceType. More often than not, we expect the two types to be the same, although, it is certainly possible to define iterators that have different distance and difference types.

The remainder of the concept defines the semantics of addition and subtraction of random access iterators and pointers—essentially pointer arithmetic—in terms of the DifferenceType. The += operator, for example, is equal to the application of advance and it returns a reference to the advanced iterator. The precondition is_valid(advance(i, n)) implies that [i, i + n) is, minimally, a weak range. Remember that is_valid guarantees the preconditions of the expression are met.

Two axioms state the commutativity and associativity of expressions involving random access iterators and distance values. The next three axioms give a Peano-like formulation of the meaning of this arithmetic. That is:

- 0 is the additive identity. Adding 0 to an iterator does not advance the iterator.
- Addition can be defined recursively as a sequence of increments.
- The successor of a random access iterator is distinct from its predecessor.

This differs from Peano arithmetic in one key notion: RandomAccessIterators do not have a fixed zero element. This means that one cannot inductively reason about an arbitrary iterator without stating the bounds of the range to which it belongs. This is why all STL algorithms take bounded and weak iterator ranges.

3.6 Rearrangements

There are several additional iterator concepts that are commonly applied to families of algorithms. These are the so-called *rearrangement* concepts. They group together iterator requirements of algorithm families. There are 6 relational concepts for rearrangements:

```
concept Permutable<ForwardIterator I>
concept Mergeable<InputIterator I1, InputIterator I2, Incrementable Out>
concept Mergeable<InputIterator I1, InputIterator I2,  Incrementable Out, typename R>
concept Sortable<ForwardIterator I>
concept Sortable<ForwardIterator I, typename R>
```

The Permutable concept describes a requirement for permuting or rearranging the elements of an iterator range.

```
concept Permutable<ForwardIterator I> =
  Semiregular<ValueType<I>> &&
  IndirectlyMovable<I, I>;
```

Rearrangement is achieved simply by exchanging underlying values. The IndirectlyMovable requirement allows Permutable iterators to directly exchange values within a range. We also require the ValueType to be Semiregular. This guarantees that permutation algorithms can declare and use temporaries as needed.

Admittedly, the requirement on Semiregular is too strong; it unnecessarily over-constrains permutation algorithms by requiring copy semantics. For example, this should be a valid program:

```
vector<unique_ptr<T>> v = { ... };
reverse(v.begin(), v.end());
```

The program will not compile using the current concept design because we require Semiregular for Permutable iterators. We have opted to present this stricter design because the alternative would require additional concepts specifically for copy and move semantics. We present that design in Appendix D.

The Mergeable concepts describe the requirements of algorithms that merge sorted sequences into an output sequence. The first overload requires that the underlying value types are TotallyOrdered, and the second is defined in terms of a generalized Relation (actually a strict weak order).

```
concept Mergeable<InputIterator I1,
                  InputIterator I2,
                  WeaklyIncrementable Out> =
  TotallyOrdered<ValueType<I1>, ValueType<I2>> &&
  IndirectlyCopyable<I1, Out> &&
  IndirectlyCopyable<I2, Out>;

concept Mergeable<InputIterator I1,
                  InputIterator I2,
                  WeaklyIncrementable2 Out,
                  Relation<ValueType<I1>, ValueType<I2>> R> =
  IndirectlyCopyable<I1, Out>;
```

The Sortable concepts describe the common requirements of algorithms that permute sequences of iterators into an ordered sequence (e.g., sort). The first overload requires the underlying value type to be TotallyOrdered, and the second is generalized over a (strict weak order) Relation.

```
concept Sortable<ForwardIterator I> =
  TotallyOrdered<ValueType<I>> && Permutable<I>;

concept Sortable<ForwardIterator I, Relation<ValueType<I>> R> =
  Permutable<I>;
```

Grouping the common requirements of an algorithmic family produces much cleaner requirements than if we tried to constrain each algorithm individually. An added benefit of this is that it allows us to conceptually group algorithms by their requirements; this will help programmers understand the inherent relationships between different kinds of algorithms that have similar requirements.

3.7 Standard Iterators

This iterator design differs from standard iterators primarily in two ways:

1. there are more iterator concepts, and

2. There is no explicit output iterator concept.

We consider the larger number of concepts an artifact of three design requirements. First, we have iterator properties like Readable, MoveWritable, and Writable that capture common requirements for more advanced concepts and algorithms. Second, we defined concepts to represent actual use in algorithms. There are a large number of algorithms where iterators are not compared for equality. This observation led to the creation of WeaklyIncrementable and WeakInputIterator. Third, we encapsulated common iterator requirements for families of algorithms. Rearrangement concepts like Permutable and Sortable are useful for expressing requirements very succinctly and are a vast improvement over the C++0x proposals (C++ Standards Committee, 2009).

```
concept WeakOutputIterator<WeaklyIncrementable Out, typename T> =
  Writable<Out, T>;

concept OutputIterator<WeaklyIncrementable Out, typename T> =
  EqualityComparable<Out> && Writable<Out, T>;
```

This specification of output iterators parallels input iterators. WeakOutputIterator is used in a large number of algorithms. OutputIterator is used in exactly three: fill, generate, and iota. The requirement for fill would look like this if we used OutputIterator in the design:

```
template<WeaklyIncrementable Out, typename T>
  requires OutputIterator<Out, T>
void fill(Out first, Out last, const T& value);
```

On one hand, this could be seen as a simpler specification of requirements. On the other hand, there are a number of redundant requirements stated between the template argument type (WeaklyIncrementable<I>) and the requires clause OutputIterator<Out, T>.

Finally, we would also have to create similar output iterator concepts for move-based algorithms such as move and move_backward. Obviously, the conclusion we reached was that no OutputIterator concept was needed at this time. In the future, we may find ways to express output iterator requirements more succinctly and re-introduce the concept as needed.

3.8 Random Number Generators

There are three algorithms in the STL that require random number generators: two overloads of random_shuffle, and shuffle. These algorithms contain two designs for random number generators. The first, the RandomNumberGenerator, is a holdover from the original STL. The second is part of the larger C++11 random number library (C++ Standard, rand).

A RandomNumberGenerator is a unary Function that takes a positive integral value, n, and returns a uniformly distributed random number in the range [0, n).

```
concept RandomNumberGenerator<typename Gen, Integral N> =
  Function<Gen, N> &&
  Convertible<ResultType<Gen, N>, N> &&
  axiom (Gen gen, N n) {
    m = gen(n), 0 >= m && m < n;
  };
```

Unfortunately, our ability to make axiomatic statements about the probability distribution of Gen is limited by the fact that we don't have sufficient syntax to say, "for an unlimited number of trials, the frequency with which the value m is observed approaches 1.0/n."

A UniformRandomNumberGenerator is a nullary Function that generates uniformly distributed values of some unsigned integer type.

```
concept UniformRandomNumberGenerator<Function Gen> =
  UnsignedIntegral<ResultType<Gen>>;
```

As with the previous concept, the syntax required to state properties of the generators' uniformity and periodicity is currently beyond our abilities. C++11 establishes a more comprehensive set of requirements for random number generators including seed sequences and random distributions. The requirements of STL algorithms did not necessitate the specification of concepts for these requirements. Doing so should be a straightforward task.

4 Conclusions

The purpose of this report is to describe the requirements of the STL algorithms, the concepts used to state those requirements, and the language features used to do both. The concept design was subject to a number of requirements. In particular, we wanted to state template requirements tersely and clearly so that it would be obvious what was required. We also wanted a design with few, semantically meaningful concepts. Finally, we wanted the requirements to be compatible with those in C++11.

While we think we have achieved the first two goals, this design is not fully compatible with C++11. We deviate from the standard requirements in a number of algorithms by "fixing" some template parameters as an iterator's associated type. In other algorithms, our stricter interpretation of cross-type relations and operations excludes instantiations where the semantics of the operation have not been formally described. Finally, all Permutable algorithms are over-constrained by a Semiregular requirement. This means that many algorithms that would be compatible for non-copyable types (e.g. reverse) are not in this design. We address this in Appendix D.

The language used to express these concepts is conservatively designed and fairly minimal in its specification. In particular, we did not want the language to impose any programming model that would dramatically alter the way that people write and use libraries. The language features presented in this work emphasize template argument checking. There is still much work to be done extending these features to address e.g. separate type checking.

4.1 Outstanding Issues

This work is clearly a starting point. There are a number of issues remaining to be addressed in this report. In particular, we think we can further simplify the specification of template constraints, and we would like to support preconditions and postconditions.

4.1.1 Simplifying Algorithm Requirements

One outstanding problem is the redundancy of parameter declarations for families of generic algorithms. For example, consider the family of set operations described in §2.3.4. Each algorithm has three template parameters (or four if you provide a Relation) that have the exact same requirements. We could dramatically simplify the declaration of these algorithms if we could use the Mergeable concept to introduce the algorithm's parameters. One speculative solution is:

```
template<Mergeable M>
M::Out set_union(M::I1 first1, M::I1 last1, M::I2 first2, M::I2 last2, M::Out result);
```

M acts as an alias to the template parameters of the Mergeable concept and the type names are qualified by that alias. Obviously, this syntax doesn't distinguish between the two overloads of M, and may have some ambiguity (does M refer to the parameters of Mergeable or is M a Mergeable type?)

We think that this is an important problem to address. Failing to provide syntax to adequately address these issues will result in the use of macros to solve the problem. This is, in many ways, directly analogous to the use of macros to simplify the declaration of template parameter lists for out-of-line member functions.

4.1.2 Preconditions and Postconditions

The axioms presented in this report are not sufficient to fully reason about the behavior of a program. While they do help describe the meaning of an expression, they need to be combined with preconditions and postconditions to support a broader range of static checking applications.

The exact form that preconditions and postconditions should take will be, we are sure, a matter of great debate. However, we know from experience that they must not be simple runtime assertions that can be turned off with a compiler flag. This model is simplistic and does not account, for properties that cannot be evaluated at runtime (e.g. universally quantified properties).

Appendix B speculates on some aspects of preconditions and postconditions, particularly the definition of *properties* which can be used to state un-evaluable requirements on function arguments.

4.2 Future Work

We plan to continue developing conceptual descriptions of the generic libraries to help refine our concept design and the language used to describe those concepts. In particular, we plan to develop requirements and concepts for other components of the C++ Standard Library: numeric algorithms (an early draft of that work was actually removed from this report) and data structures, containers, I/O streams, strings, random number generators, etc.

At the same time, we plan to build a compiler (based on Clang) for the emerging language. This prototype will support the full range features described in this report. We hope to have an initial version ready by early February, 2012.

Acknowledgments

We thank A9.com for hosting this meeting. This work was partly supported by grants from the National Science Foundation (CCF-0702717, A3350-32525-CS, and A0040-32525-CS), the Lilly Endowment, Inc. (2008 1639—000), and King Abdullah University of Science and Technology (KUS-C1-016-04).

Marcin Zalewski and Larisse Voufo directly participated in editing of this report, and Jeremiah Willcock participated in discussions and provided many useful comments.

We would also like to thank Paul McJones, Alex Stepanov, Dan Rose, Anil Gangolli, Abe Skolnik and Michael Lopez for their revisions, comments, and additional discussion about the topics presented in this report.

Appendices

Appendix A Summary of the Language

In this appendix, we give a summary of the language features used to define the concepts for and constrain the algorithms of the STL. A more formal specification of the language and implementation issues currently under investigation.

A.1 Language Description

The grammar defined here is based on the C++11 Standard (C++ Standards Committee, 2011). Grammarical productions and descriptions thereof that are not defined in this appendix can be found in there.

A.1.1 Concept Definitions

A concept definition is a declaration (C++ Standard, dcl.dcl):

declaration:
    block-declaration
    function-definition
    template-declaration
    ...
    concept-definition

A concept definition starts with the concept keywrd and is followed by its name, a template parameter list, and its body (a conjunction of requirements and Boolean expressions).

concept-definition:
    `concept` *concept-name* `<` *template-parameter-list* `>` `=` *concept-body* `;`

We had debated whether or not concepts could be forward-declared: introducing the concept name without providing a complete definition of the concept. Since we did not need that feature to describe the concepts for the STL, we did design the grammar to support the feature.

A.1.2 Concept Names

A concept name is a simple identifier (C++ Standard, lex.name), and a concept id refers to a concept specialiazation (similar to a template-id (C++ Standard, temp.names)).

concept-name:
    identifier

concept-id:
    concept-name `<` *template-argument-list* `>`

A *concept-id* denotes the evaluation of concept's requirements on the given arguments.

A *concept-id* can be used as a primary expression in C++ (it is an *unqualified-id*). When appearing in an expression, it is evaluated as the booolean constant true of false, depending on whether its template arguments satisfy its stated requirements. Technically, that makes the following a legal program:

```
if(Regular<T>) {
  // do something
}
```

This does *not* define a "static if". The compiler will still parse and analyze the nested statements in the if block, regardless of whether Regular<T> is true or false.

A.1.3 Constrained Template Parameters

A concept's template parameter specification is the same as that of a template declaration. However, we have extended the usual template parameter production (C++ Standard, temp.param) with constrained template parameters:

template-parameter:
    ...
    constrained-template-parameter

constrained-template-parameter:
    concept-id ...$_{opt}$ *identifier constrained-default-argument*$_{opt}$ *concept-name* ...$_{opt}$ *identifier constrained-default-argument*$_{opt}$

constrained-default-argument:
    = *type-id*
    = *assignment-expression*
    = *id-expression*

A constrained template parameter uses a concept name or id (described before) as the "type" of the declared template parameter, a shorthand for writing a requirement involving the concept and the parameter. For example:

```
concept A<typename T> = ...;
concept B<typename T> = ...;

concept C1<A T, B U> = X<T>;
concept C2<typename T, typename U> = A<T> && B<U> && X<T>;
```

C1 and C2 have equivalent requirements.

A *concept-id* can be used if the referenced concept takes multiple template arguments. The declared template parameter is substituted as the first argument of the concept. For example:

```
concept A<typename T, typename U> = ...;

concept C1<typename T, A<T> U> = X<T>;
concept C2<typename T, typename U> = A<U, T> && X<T>;
```

Again, C1 and C2 have equivalent requirements.

The kind of constrained template parameter (type parameter, non-type parameter, or template template parameter) is deduced from the kind of first template parameter of the constraining template. For example:

```
concept True<bool B> = b;

concept C<True B> = ...;

C<true> // okay
C<bool> // error: bool is a type
```

The first argument of C has the constraint True, and its first template parameter is the non-type parameter with type bool.

Template parameters can be assigned a default type or value. Example:

```
concept C1<Regular T, Regular U = T>;
```

A constrained template parameter may also indicate a template parameter pack. For example:

```
concept NumberList<Integral... Args> = ...;
```

This has the same meaning as:

```
concept NumberList<typename... Args> = Integral<Args>...;
```

Again, the type of parameter pack (type, non-type, template) is derived from the first template parameter of the constraining concept (Integral). The precise meaning of this expansion is described below.

A.1.4 Concept Bodies

A concept's body consists of a conjunction of one or more concept clauses. The conjunction of clauses is represented with the usual &&:

concept-body:
    concept-requirement-seq

concept-requirement-seq:
    concept-requirement `&&` *concept-requirement-seq*
    concept-requirement

concept-requirement:
    constant-expression
    concept-id
    requires-block
    axiom-block

1. a constant expression whose result is convertible to bool,
2. a *concept-id*, representing the use of another concept,
3. a requires block expressing syntactic requirements, or
4. an axiom block representing semantic requirements.

Constant expressions can include the evaluation of type traits, such as is_same<T, U>::value from the definition of Same in §3.2.1).

A concept is true when all of its requirements evaluate to true.

A.1.5 Syntactic Requirements

A requires block can contain one or more of syntactic and semantic requirements and can be optionally given a descriptive name:

requires-block:
    `requires` $identifier_{opt}$ `(`$_{opt}$ $parameter\text{-}declaration\text{-}list_{opt}$ `)`$_{opt}$ `{` *requirement-seq* `}`

Examples of three different in which requirement blocks can be written are:

```
concept C1<typename T> = requires { ... };
concept C2<typename T> = requires (T a, T b) { ... };
concept C3<typename T> = requires my_requirement(T a, T b) { ... };
```

A requires block can be parameterized to introduce objects for writing requirements, and they can be optionally named to help break up long lists of requirements. Note that the name of a requirement has no meaning within the program; it does not denote a declaration.

A requires block consists of a sequence of requirement statements.

requirement-stmt-seq:
    requirement-stmt
    requirement-stmt-seq requirement-stmt

The requirements are specified in a sequential fashion, although their ordering has no effect on the overall meaning. A requirement eiter specifies a valid expression, an associated type, or an axiom describing the semantics of those required expressions and types.

requirement-stmt:
    valid-expr-requirement ;
    assoc-type-requirement ;
    axiom-block

A valid expression describes syntax representing the set of valid operations on the concept's template arguments. They are specified in one of four ways:

valid-expr-requirement:
    expression
    type-id { *expression* }
    type-id = { *expression* }
    type-id == { *expression* }

The first form describes syntax with an unconstrained result type. For example, the required syntax `*o = value` in `Writable` is unconstrained. The three remaining productions describe requirements on the result type of a required expression. Note that the syntax for stating result type constraints is a slightly modified version of declaration and initializer syntax (C++ Standard, dcl.decl, dcl.init). Examples of the actual requirements corresponding to the different syntaxes are:

```
bool { a < b };    // Explicit conversion
C = { op(a, b) };  // Implicit conversion
T* == { &a };      // Same type requirement
```

For the first two conversion requirements, the corresponding we could check the the validity of the following declarations:

```
bool b{ a < b };
C c = { op(a, b) };
```

Simply inserting a variable name after the type causes the syntax to be a (possibly) valid declaration. This is exactly how these requirements are checked. Note that we could equivalently have written `Convertible<decltype(op(a, b)), C>` as the type constraint on the second expression. The same-type requirement is interpreted in exactly this manner:

```
T* p = { &a };
Same<decltype(&a), T*>;
```

The result of the expression must be the same as the type of the declarator.

An associated type requirement declares a requirement for a type function or template alias:

associated-type-requirement:
    simple-template-id ;

For example:

```
concept Iterator<typename I> =
  requires {
    IteratorCategory<I>;  // must refer to a valid type
  } && ...
```

A.1.6 Semantic Requirements

An axiom is a block of statements describing the properties of a required expression. Like syntactic requirements, axioms can be optionally named and may introduce formal parameters.

axiom-block:
    `axiom` *identifier*$_{opt}$ `(` *parameter-declaration-list*$_{opt}$ `)` `{` *axiom-seq* `}`

axiom-seq:
    axiom-seq axiom
    axiom

Note that the syntax presented here is restricted to the uses in the report. We have not tried to defined a more general specification language.

Axioms consist of sequences of Boolean expressions of the following forms:

axiom:
    expression-statement
    alias-declaration

The set of statements that can be written in an axiom block is a restricted set of what might be written in a function body. In particular, we restrict this to expression statements and alias declarations. We have not found any need to write if or while statements in axioms.

In general, axioms are comprised of C++ expressions written as equations. A common use is to define equations over equal values, as in:

```
axiom commutative(T a, T b) {
  a + b == b + a;
}
```

Many axioms presented in this report rely on two additional operators: => and <=>. We introduce these as extensions of the standard set of C++ operators. Note that while axioms are not evaluated as part of a program's execution, we do describe how these operators are be evaluated (they may be useful for test cases).

The implication operator (=>) has the following syntax:

implication-expression:
    logical-or-expression logical-or-expression => *implication-expression*

It groups right-to-left. When evaluated, both operands are contextually converted to bool and the result is true when either both operand are true or the first operand is false. Like the && and || operators, the => guarantees left-to-right evaluation, and the second operand is not evaluated if the first operand is false. Examples include the symmetric and transitive properties of equality (as given in the EqualityComparable concept).

```
a == b => b == a;
a == b && b == c => a == c;
```

The syntax of the logical equivalence <=> operator is:

equivalence-expression:
    implication-expression
    equivalence-expression <=> *implication-expression*

It groups left-to-right. When evaluated, both operators are contextually converted to bool and the result is true only when both of its operands are true. For example:

```
a != b <=> !(a == b);
```

It is not intended that the implication and equiavalence operators be overloaded. They are meant to define primitive relations Boolean expressions.

To accommodate the new operands, we modify the specification of the conditional operator:

conditional-expression:
    equivalence-expression
    equivalence-expression `?` *expression* `:` *assignment-expression*

Because *concept-id*s are evaluated as boolean expressions, we can use them as arguments in an implication. For example, the axiom of the Writable concept includes the following statement:

```
Readable<Out> && Same<ValueType<Out>, T> =>
  is_valid(*o = value) => (*o = value, eq(*o, other));
```

This axiom states that, if the template parameter Out is Readable and its value type is the same as the type written to Out, writing a value through an output iterator ensures that a subsequent read will yield an equal value. The Readable and Same contexts locally constrain the syntax of the second operand—it lets write the expression eq(*o, other).

Alias declarations allow us to simplify some requirements by creating type aliases. For example:

```
axiom (T t, U u) {
  using C = CommonType<T, U>;
  t + u == C{t} + C{u};
}
```

Here, we have introduced C so we don't have to write the longer CommonType<T, U>.

Axioms can be either independent concept clauses, or they can be embedded in requirement clauses. The choice has no effect on the meaning of the program. For examples, axioms about subtraction might be written as:

```
concept X<Number T> =
  requires (T a) {
    T { a - a };
    axiom { a - a == T{0}; }
  } &&
  axiom (T a, T b) {
    a - b == a + -b;
  };
```

Throughout this report, we have written axioms near to the syntax they describe.

A.1.7 Constrained Templates

One of the main objectives of a concept language is the ability to constrain template arguments of generic algorithms. We extend the syntax of template declarations to allow specifying constraints:

template-declaration:
    `template` `<` *template-parameter-list* `>` *requires-clause*$_{opt}$ *declaration*

requires-clause:
    `requires` *logical-and-expression*

We extend *template-declaration* template declarations with an optional *requires-clause* (indicated by the requires keyword). The requires clause is a conjunction of expressions (usually *concept-ids*). Note that the expression must be a constant expression whose result is convertible to bool.

We also allow the same shorthand syntax for constraints that we used in concepts definitions. For example, the two following declarations are equivalent:

```
template<SomeConcept T1, AnotherConcept<T1> T2>
f(T1 a, T2 b);

template<typename T1, typename T2>
requires SomeConcept<T1> && AnotherConcept<T2, T1>
f(T1 a, T2 b);
```

Because concepts can and often are defined in terms of other concepts, one can write constraints in different ways. Our design does not preclude redundancy in constraints, and, in fact, our convention requires it in some cases. We have chosen to avoid "naked" template parameters in signatures by specifying the "strongest" (most specific) concept for every template parameter where possible. For example, in §2.3.1 we write:

```
template<RandomAccessIterator I, Relation<ValueType<I>> R>
requires Sortable<I, R>
void nth_element(I first, I nth, I last, R comp);
```

Sortable already requires Relation for the R parameter, and, strictly speaking, the requirement Relation<ValueType<I>> R is not necessary, but we prefer it to typename R.

A.2 Grammar Summary

In this section we list all the productions of the grammar in one place for easy reference and comprehension.

declaration:
- *block-declaration*
- *function-definition*
- *template-declaration*
- *...*
- *concept-definition*

concept-definition:
- `concept` *concept-name* `<` *template-parameter-list* `>` `=` *concept-body* `;`

concept-name:
- *identifier*

concept-id:
- *concept-name* `<` *template-argument-list* `>`

template-parameter:
- *...*
- *constrained-template-parameter*

constrained-template-parameter:
- *concept-id* $\ldots_{opt}$ *identifier* *constrained-default-argument*$_{opt}$ *concept-name* $\ldots_{opt}$ *identifier* *constrained-default-argument*$_{opt}$

constrained-default-argument:
- `=` *type-id*
- `=` *assignment-expression*
- `=` *id-expression*

concept-body:
- *concept-requirement-seq*

concept-requirement-seq:
- *concept-requirement* `&&` *concept-requirement-seq*
- *concept-requirement*

concept-requirement:
- *constant-expression*

concept-id
requires-block
axiom-block

requires-block:
`requires` *identifier*$_{opt}$`(`$_{opt}$ *parameter-declaration-list*$_{opt}$`)`$_{opt}$`{` *requirement-seq* `}`

requirement-stmt-seq:
requirement-stmt
requirement-stmt-seq requirement-stmt

requirement-stmt:
valid-expr-requirement `;`
assoc-type-requirement `;`
axiom-block

valid-expr-requirement:
expression
type-id `{` *expression* `}`
type-id `=` `{` *expression* `}`
type-id `==` `{` *expression* `}`

associated-type-requirement:
simple-template-id `;`

axiom-block:
`axiom` *identifier*$_{opt}$`(` *parameter-declaration-list*$_{opt}$`)` `{` *axiom-seq* `}`

axiom-seq:
axiom-seq axiom
axiom

axiom:
expression-statement
alias-declaration

implication-expression:
logical-or-expression logical-or-expression `=>` *implication-expression*

equivalence-expression:
implication-expression
equivalence-expression `<=>` *implication-expression*

conditional-expression:
equivalence-expression
equivalence-expression `?` *expression* `:` *assignment-expression*

template-declaration:
`template` `<` *template-parameter-list* `>` *requires-clause*$_{opt}$ *declaration*

requires-clause:
`requires` *logical-and-expression*

Appendix B Preconditions and Postconditions

This appendix describes the preconditions and postconditions of the algorithms in the STL. Obviously, C++ does not provide sufficient notation to support the task so we have taken some liberties to extend C++ in ways that will. The language features included in this section are purely speculative; we do not mean to include them as part of the C++ concept proposal. We include them here in order to be precise about the meaning of the algorithms in Section §2.

The preconditions and postconditions for the STL algorithms are heavily motivated by the *Elements of Programming* book by Stepanov and McJones. Many of the preconditions in that book were stated as *properties*, which define the mathematical properties of objects using existential and universal quantification, implication, logical equivalence, etc.

Much of that syntax is already used in the axioms defined by concepts. In order to express these properties in a C++-like language, we allow axioms to be defined outside of concepts. In particular we introduce some additional notation to express universal and existential quantification.

Throughout this appendix, we refer to algorithms and supporting data structures that are not defined in the STL. We do so to simplify the specification of many pre- and post-conditions. Prototype implementations of these supplemental algorithms and data structures can be found in the Origin library Sutton (2011).

B.1 Property Library

The property library defines a number of basic properties used throughout this appendix to specify preconditions and postconditions.

B.1.1 Iterator Ranges

In particular, it defines the meaning of *weak ranges*, *counted ranges*, and *bounded ranges* and the validity of operations on iterators in those ranges.

A *weak range* is defined by an iterator that can be incremented n times. It is described by the following property:

```
template<WeaklyIncrementable I>
property is_weak_range(I first, DistanceType<I> n)
{
  for all(DistanceType<I> i)
    (0 <= i && i <= n) => is_valid(next(first, i));
}
```

This is similar any other function template except that a) it is introduced by the keyword property, and b) it has no specified result type. The result of a property is always bool.

The **for all** construct is the universal quantifier, ∀ (the identifier all is assumed to be contextual keyword). The quantifier introduces a variable i that ranges over all values of DistanceType<I>. Like a usual for loop, the quantifier has a nested statement describing the property to be satisfied: an implication in this case. The is_valid predicate is the same as used to specify iterator axioms. Here, is_valid asserts the validity of its expression argument. The result of a quantifier is a bool value, which is hypothetically true when the property is satisfied for all values of i.

Like a concept's axioms, properties are not intended to be evaluated at runtime. Whereas axioms are quantified over their parameters and assumed to be true for all values (applicable) values of their corresponding types, a property describes a constraint on *specific* objects, namely the arguments to which it is applied. In total, the axiom states that advancing from first to each value of i in the specified range must not result in undefined behavior. For example, if we could write:

```
int A[] = {0, 1, 2, 3 }
// is_weak_range(a, 1); // is true
// is_weak_range(a, 10); // is false
```

Note that if i is a valid iterator (i.e. points to an object), then is_weak_range(i, 1) is true. You can always increment a valid iterator at least once.

A *counted range* is a weak range that has no loops in the orbit of its increment operator. The specification of this property is:

```
template<WeaklyIncrementable I>
property is_counted_range(I first, DistanceType<I> n)
{
  weak_range(first, n) &&
  for all(DistanceType<I> i)
    for all(DistanceType<I> j)
      (0 <= i && i < j && j <= n) => next(first, i) != next(first, j);
}
```

Because quantifiers are expressions returning bool, they can be used in Boolean expressions. This means that they can also be mixed with more conventional predicates like is_sorted.

A *bounded range* is a counted range over the distance from first to last.

```
template<WeaklyIncrementable I>
requires EqualityComparable<I>
property is_bounded_range(I first, I last)
{
  for some(DistanceType<I> n)
    is_counted_range(first, n) && next(first, n) == last;
}
```

The quantifier **for some** is the existential quantifier, $\exists$.

A readable_range is a weak, counted, or bounded range where dereferencing is a valid operation. We commonly refer to readable ranges as *input ranges*.

```
template<WeakInputIterator I>
property is_readable_range(I first, DistanceType<I> n)
{
  is_weak_range(first, n) &&
    for all(I i : range(first, n))
      is_valid(*i);
}

template<InputIterator I>
property is_readable_range(I first, I last)
{
  is_bounded_range(first, last) &&
    for all(I i : range(first, last))
      is_valid(*i);
}
```

We have overloaded the property based on the input types of the algorithm. The first overload states the readability for weak ranges and the second for bounded ranges. If an algorithm requires a readable *counted* range, then the preconditions should state an additional requirement on the is_counted_range. This design is slightly different than that used in the EoP book. Because counted ranges are not frequently used preconditions, we have tried to simplify the properties for weak and bounded ranges.

The quantifiers in these axioms are bound to specific values. The range function returns a range object over which the variable is quantified. The two overloads of range are:

1. range(first, n) — returns the counted range [first, first + n).

2. range(first, last) — returns the bounded range [first, last).

This is not intended to be a hypothetical facility. The implementation of range and its corresponding range types are straightforward and can be found in Origin.

The readable property establish the validity of the dereferencing operator for iterators in the given ranges. Specifically, this says that the iterators in these ranges are readable everywhere except the limit. That is, for example, if first == last, the *first is not a valid operation.

A *movable range* and *writable range* are sequence of iterators to which a value can be moved or assigned, respectively. These are more generally referred to as *output ranges*. There are two definitions for each:

```
template<WeaklyIncrementable I, typename T>
requires Writable<T, I>
property is_movable_range(I first, DistanceType<I> n, T x)
{
  is_weak_range(first, n) &&
    for all(auto i : range(first, n)
      is_valid(*i = move(x));
}

template<WeaklyIncrementable I, typename T>
requires EqualityComparable<I> && Writable<T, I>
property is_movable_range(I first, I last, T x)
{
  is_bounded_range(first, last) &&
    for(I i : range(first, last))
      is_valid(*i = move(x));
}

template<WeaklyIncrementable I, typename T>
requires Writable<T, I>
property is_writeable_range(I first, DistanceType<I> n, T x)
{
  is_movable_range(first, n) &&
    for all(auto i : range(first, n)
      is_valid(*i = x);
}

template<WeaklyIncrementable I, typename T>
requires EqualityComparable<I> && Writable<T, I>
property is_writable_range(I first, I last, T x)
{
  is_movable_range(first, last) &&
    for(I i : range(first, last))
      is_valid(*i = x);
}
```

Here, we state the move or assignment in terms of an actual object, x, that is being assigned. This is necessarily different from the similar definitions in the EoP book where value writable iterators have a built-in notion of value type. Again, we override the predicate for weak and bounded ranges. Algorithms requiring counted writable ranges will use the overload for weak ranges and state a second requirement of is_counted_range.

We define *permutable ranges* as ranges whose values can be exchanged and written to (or overwritten).

```
template<Permutable I>
property is_permutable_range(I first, DistanceType<I> n)
{
  is_readable_range(first, n) && is_movable_range(first, n, *first);
}

template<Permutable I>
property is_permutable_range(I first, I last)
{
  is_readable_range(first, last) && is_movable_range(first, last, *first);
}
```

It is often useful to assert that an iterator can be found within a range. There are two properties used throughout this report to ensure that:

```
template<WeaklyIncrementable I>
requires EqualityComparable<I>
property in_range(I i, I first, I last)
{
  is_bounded_range(first, i) && is_bounded_range(i, last);
}

template<WeaklyIncrementable I>
requires EqualityComparable<I>
property in_closed_range(I i, I first, I last)
{
  in_range(first, last, i) || i == last;
}
```

The in_range property can be used to ensure that the iterator i can be found in [first, last). The in_closed_range property ensures that i is in the closed range [first, last].

B.1.2 Relations

There are a number of properties that can be associated with Relations.

A relation is *reflexive* if, for all a, a is related to itself.

```
template<typename R>
property reflexive(R r)
{
  for all(DomainType<R> a) r(a, a);
}
```

This property is unconstrained in order to simplify its usage. We want to write, reflexive(r) in order to assert the property. The DomainType alias is just a placeholder for the relation's domain type (which not actually be deducable—R could be polymorphic). The intent is to document the requirement rather than implement a testable program.

The opposite of a reflexive relation is an *irreflexive* relation. For every a, a is not related to itself.

```
template<typename R>
property irreflexive(R r)
{
  for all(DomainType<R> a) !r(a, a);
}
```

A relation is *symmetric* if, for all a and b, a is related to b then b is also related to a.

```
template<typename R>
property symmetric(R r)
{
  for all(DomainType<R> a, b) r(a, b) => r(b, a);
}
```

As a shorthand, we allow multiple quantified variables of the same type to be introduced in a single **for all** declaration. This is only allowed when the variables range over all values of the type.

The opposite of a symmetric relation is an *asymmetric* relation. For all a and b, if a is related to b, then b is not related to a.

```
template<typename R>
property asymmetric(R r)
{
  for all(DomainType<R> a, b) r(a, b) => !r(b, a);
}
```

A relation is *transitive* if, for all a, b, and c, if a is related to b, and b is related to c, then a is also related to c.

```
template<typename R>
property transitive(R r)
{
  for all(DomainType<R> a, b, c) r(a, b) && r(b, c) => r(a, c);
}
```

Meaningful relations can be constructed from these basic properties. An *equivalence relation* is reflexive, symmetric, and transitive.

```
template<typename R>
property equivalence_relation(R r)
{
  reflexive(r) && symmetric(r) && transitive(r);
}
```

A *strict weak ordering* is a Relation that generalizes a total ordering. Formally, we can define the property as:

```
template<typename R>
property strict_weak_ordering(R r)
{
  irreflexive(r) && asymmetric(r) && transitive(r)
    && equivalence_relation(symmetric_complement(r));
}
```

The symmetric_complement function returns a function object that evaluates the expression:

```
!r(a, b) && !r(b, a)
```

for all a and b in the domain of r.

B.2 Non-Modifying Sequential Algorithms

This section describes the preconditions and postconditions of all of the non-modifying STL algorithms. We explain new syntax and properties as needed.

B.2.1 All of

```
template<InputIterator I, Predicate<ValueType<I>> P>
bool all_of(I first, I last, P pred);
```

Requires: is_readable_range(first, last)

Ensures: **for all**(I i : range(first, last)) pred(∗i).

Complexity: pred is applied at most last - first times.

B.2.2 Any of

```
template<InputIterator I, Predicate<ValueType<I>> P>
bool any_of(I first, I last, P pred);
```

Requires: is_readable_range(first, last)

Ensures: **for some**(I i : range(first, last)) pred(∗i).

Complexity: pred is applied at most last - first times.

B.2.3 None of

```
template<InputIterator I, Predicate<ValueType<I>> P>
bool none_of(I first, I last, P pred);
```

Requires: is_readable_range(first, last)

Ensures: **for all**(I i : range(first, last)) !pred(∗i).

Complexity: pred is applied at most last - first times.

B.2.4 For Each

```
template<InputIterator I, Semiregular F>
requires Function<F, ValueType<I>>
F for_each(I first, I last, F f)
```

Requires: is_readable_range(first, last)

Ensures: Let result = for_each(first, last, f) where

- first == last => eq(result, f), otherwise
- eq(result, (f(∗first), for_each(first + 1, last, f))).

Complexity: f is applied exactly last - first times.

B.2.5 Find

```
template<InputIterator I, EqualityComparable<ValueType<I>> T>
I find(I first, I last, const T& value)

template<InputIterator I, Predicate<ValueType<I>> P>
I find_if(I first, I last, P pred)

template<InputIterator I, Predicate<ValueType<I>> P>
I find_if_not(I first, I last, P pred)
```

Requires: is_readable_range(first, last)

Ensures: For find, let i = find(first, last, value) where

- in_closed_range(i, first, last), and
- none_equal(first, i, value), and
- i != last => *i == value.

Ensures: For find_if, let i = find_if(first, last, pred) where

- in_closed_range(i, first, last), and
- none_of(first, i, pred), and
- i != last => pred(*i) == true.

Ensures: For find_if_not, let i = find_if_not(first, last, pred) where

- in_closed_range(i, first, last), and
- all_of(first, i, pred), and
- i != last => pred(*i) == false.

Complexity: The == operator or corresponding pred function is applied at most last - first times.

The none_equal algorithm is not present in the STL. It is equivalent to writing:

```
none_of(first, last, [&](const ValueType<I>& x) {return x == value;});
```

We find it easier to write pre- and post-conditions in terms of other algorithms. We also assume the existence of all_equal and any_equal, having similar definitions.

B.2.6 Find First

```
template<InputIterator I1, ForwardIterator I2>
requires EqualityComparable<ValueType<I1>, ValueType<I2>>
I1 find_first_of(I1 first1, I1 last1, I2 first2, I2 last2);

template<InputIterator I1,
         ForwardIterator I2,
         Predicate<P, ValueType<I1>, ValueType<I2>> P>
I1 find_first_of(I1 first1, I1 last1, I2 first2, I2 last2, P pred)
```

Requires:

- is_readable_range(first1, last1), and
- is_readable_range(first2, last2).

Ensures: For the first overload, let i = find_first_of(first1, last1, first2, last2) where

- in_closed_range(i, first1, last1), and
- **for all**(I1 j : range(first1, i)) none_equal(first2, last2, *j), and
- i != last1 => any_equal(first2, last2, *i).

For the second overload Let i = find_first_of(first1, last1, first2, last2, pred) where

- in_closed_range(i, first1, last1), and
- **for all**(I1 j : range(first1, i)) none_of(first2, last2, bind(pred, *j, _1)), and
- i != last1 => any_of(first2, last2, bind(pred, *i, _1)).

Complexity: The operator == or corresponding pred is applied at most (last1 - first1) * (last2 - first2) times.

The specification of the second overload relies on the bind function. We are currying an argument of the binary pred to make it a viable argument for any_of and none_of.

B.2.7 Adjacent Find

```
template<ForwardIterator I>
requires EqualityComparable<ValueType<I>>
I adjacent_find(I first, I last)

template<ForwardIterator I, Relation<ValueType<I>> R>
I adjacent_find(I first, I last, R comp);
```

Requires: is_readable_range(first, last)

Ensures: For the first overload, let i = adjacent_find(first, last) where

- in_closed_range(i, first, last). Furthermore,
- first == last => i == last, or
- none_adjacent(first, i), and
- i != last => i + 1 != last && *i == *(i + 1).

Ensures: For the second overload, let i = adjacent_find(first, last, comp) where

- in_closed_range(i, first, last). Furthermore,
- first == last => i == last, or
- none_adjacent(first, i, comp), and
- i != last => i + 1 != last && comp(*i, *(i + 1)).

Complexity: When first != last, exactly min((i -first) + 1, (last - first) - 1) applications of == or the corresponding comp relation. Otherwise, no operations are performed.

The none_adjacent function is an auxillary predicate that returns true if and only if last - first < 2 or there are no iterators i and j in [first, last) with j == i + 1 and j != last where *i == *j.

B.2.8 Count

```
template<InputIterator I, EqualityComparable T>
requires EqualityComparable<ValueType<I>, T>
DistanceType<I> count(I first, I last, const T& value)

template<InputIterator I, Semiregular P>
requires Predicate<P, ValueType<I>>
DistanceType<I> count_if(I first, I last, P pred)
```

Requires: is_readable_range(first, last)

Ensures: For count, returns the number of elements in [first, last) that are equal to value. That is, count returns the cardinality of the multiset $\{\texttt{i} \in [\texttt{first}, \texttt{last}) : *\texttt{i} == \texttt{value}\}$.

Ensures: For count_if, returns the number of elements in [first, last) that are satisfy pred. That is, count_if returns the cardinality of the multiset, $\{\texttt{i} \in [\texttt{first}, \texttt{last}) : \texttt{pred}(*\texttt{i})\}$.

Complexity: Exactly last - first applications of == or the corresponding pred.

B.2.9 Equal and Mismatch

```
template<InputIterator I1, WeakInputIterator I2>
requires EqualityComparable<ValueType<I1>, ValueType<I2>>
bool equal(I1 first1, I1 last1, I2 first2)

template<InputIterator I1,
         WeakInputIterator I2,
         Predicate<ValueType<I1>, ValueType<I2>> P>
bool equal(I1 first1, I1 last1, I2 first2, P pred);
```

Requires:

- is_readable_range(first1, last1), and
- is_readable_range(first2, last1 - first1).

Ensures: For the first overload, **for all**(I i : range(first1, last1)) *i == *(first2 + (i - first1)).

Ensures: For the second overload, **for all**(I i : range(first1, last1)) pred(*i, *(first2 + (i - first1))).

Complexity: At most last1 - first1 applications of == or the corresponding pred function.

```
template<InputIterator I1, WeakInputIterator I2>
requires EqualityComparable<ValueType<I1>, ValueType<I2>>
pair<I1, I2> mismatch(I1 first1, I1 last1, I2 first2)

template<InputIterator I1,
         WeakInputIterator I2,
         Predicate<ValueType<I1>, ValueType<I2>> P>
pair<I1, I2> mismatch(I1 first1, I1 last1, I2 first2, P pred)
```

Requires:

- is_readable_range(first1, last1), and
- is_readable_range(first2, last1 - first1).

Ensures: For the first overload, let p = mismatch(first1, last1, first2) where

- in_closed_range(p.first, first1, last1), and
- in_closed_range(p.second, first2, first2 + (last1 - first1)), and
- equal(first1, p.first, first2), and
- p.first != last1 => *p.first != *p.second.

Ensures: For the second overload, let p = mismatch(first1, last1, first2, pred) where

- in_closed_range(p.first, first1, last1), and
- in_closed_range(p.second, first2, first2 + (last1 - first1)), and
- equal(first1, p.first, first2, pred), and
- p.first != last1 => pred(*p.first, *p.second).

Complexity: At most last1 - first1 applications of == or the corresponding pred relation.

B.2.10 Is Permutation

```
template<ForwardIterator I1, ForwardIterator I2>
requires EqualityComparable<ValueType<I1>, ValueType<I2>>
bool is_permutation(I1 first1, I1 last1, I2 first2);

template<ForwardIterator I1, ForwardIterator I2, Relation<ValueType<I1>, ValueType<I2>> R>
bool is_permutation(I1 first1, I1 last1, I2 first2, R comp);
```

Requires: Both overloads require

- is_readable_range(first1, last1), and
- is_readable_range(first2, last2 - first2).

The second overload also requires equivalence_relation(comp)

Ensures: For the first overload, **for some**(auto r : permutations(first1, last1)) equal(begin(r), end(r), first2).

Ensures: For the second overload, **for some**(auto r : permutations(first1, last1, comp)) equal(begin(r), end(r), first2, comp).

Complexity: If [first1, last1) is equal to [first2, last2) using the equal algorithm, then then exactly distance(first1, last1) applications of == or the corresponding comp relation. Otherwise, $\mathcal{O}(\backslash^{\in})$ applications where n has the value distance(first1, last1).

The function permutations returns an iterable range over all the permutations of the given sequence. This is not intended to by a hypothetical function; the next_permutation algorithm can be used to construct an iterator over permutations of a sequence. An implementation can be found in Origin.

B.2.11 Search

```
template<ForwardIterator I1, ForwardIterator I2>
requires EqualityComparable<ValueType<I1>, ValueType<I2>>
I1 search(I1 first1, I1 last1, I2 first2, I2 last2);

template<ForwardIterator I1, ForwardIterator I2, Semiregular P>
requires Predicate<P, ValueType<I1>, ValueType<I2>>
I1 search(I1 first1, I1 last1, I2 first2, I2 last2, P pred)
```

Requires:

- is_readable_range(first1, last1), and

- is_readable_range(first2, last2).

Ensures: For the first overload, let i = search(first1, last1, first2, last2) where,

- in_closed_range(i, first1, last1), and
- if i != last then
  - last1 - i >= last2 - first2, and
  - **for all**(I1 j : range(first1, i)) !equal(first2, last2, j), and
  - equal(first2, last2, i),
- otherwise there are no sub-sequences of [first1, last1) equal to [first2, last2).

Ensures: For the second overload, let i = search(first1, last1, first2, last2, pred) where,

- in_closed_range(i, first1, last1), and
- if i != last then
  - last1 - i >= last2 - first2, and
  - **for all**(I1 j : range(first1, i)) !equal(first2, last2, j, pred), and
  - equal(first2, last2, i, pred),
- otherwise there are no subsequences of [first1, last1) equivalent to [first2, last2) under pred.

Complexity: At most last1 - first1 * last2 - first2 applications of == or the corresponding pred function.

---

template<ForwardIterator I1, ForwardIterator I2>
requires EqualityComparable<ValueType<I1>, ValueType<I2>>
I1 find_end(I1 first1, I1 last1, I2 first2, I2 last2)

template<ForwardIterator I1, ForwardIterator I2, Semiregular P>
requires Predicate<P, ValueType<I1>, ValueType<I2>>
I1 find_end(I1 first1, I1 last1, I2 first2, I2 last2, P pred)

Requires:

- is_readable_range(first1, last1), and
- is_readable_range(first2, last2).

Ensures: For the first overload, let i = find_end(first1, last1, first2, last2) where

- in_closed_range(i, first1, last1), and
- if i != last then
  - last1 - i >= last2 - first2, and
  - **for all**(I1 j : range(i + 1, last1)) !equal(first2, last2, j), and
  - equal(first2, last2, i),

- otherwise there are no subsequences of [first1, last1) equal to [first2, last2)

Ensures: For the second overload, i = find_end(first1, last1, first2, last2, pred) where

- in_closed_range(i, first1, last1), and
- if i != last then
  - last1 - i >= last2 - first2, and
  - **for all**(I1 j : range(i + 1, last1)) !equal(first2, last2, j, pred), and
  - equal(first2, last2, i, pred),
- otherwise there are no subsequences of [first1, last1) equivalent to [first2, last2) under pred.

Complexity: At most last1 - first1 * last2 - first2 applications of == or the corresponding pred function.

```
template<ForwardIterator I, EqualityComparable T>
requires EqualityComparable<ValueType<I>, T>
I search_n(I first, I last, DistanceType<I> count, const T& value);

template<ForwardIterator I, Semiregular T, Semiregular P>
requires Predicate<P, ValueType<I>, T>
I search_n(I first, I last, DistanceType<I> count, const T& value, P pred);
```

Requires: is_readable_range(first, last2).

Ensures: For the first overload, let i = search_n(first, last, count, value) where

- in_closed_range(i, first, last), and
- if i != last then
  - n <= last - i, and
  - **for all**(I j : range(first, i)) !all_equal(j, j + n, value), and
  - all_equal(i, i + n, value),
- otherwise there are no sub-sequences of [first, last) whose values are all equal to value.

Ensures: For the second overload, let i = search_n(first, last, count, value, pred) where

- in_closed_range(i, first, last), and
- if i != last then
  - n <= last - i, and
  - **for all**(I j : range(first, i) !all_of(j, j + n, bind(pred, _1, value)), and
  - all_of(i, i + n, bind(pred, _1, value)),
- otherwise there are no subsequences of [first, last) whose values are all equal to value.

Complexity: At most last - first applications of == or the corresponding pred function.

B.3 Mutating Sequence Algortihms

Postconditions on output iterators are only meaningful when they are Readable and the value types of the input and output iterators EqualityComparable. If those conditions are not met, then we cannot formally specify the behavior of the algorithms.

B.3.1 Copy

```
template<InputIterator I, WeaklyIncrementable Out>
requires IndirectlyCopyable<I, Out>
Out copy(I first, I last, Out result);
```

Requires:

- is_readable_range(first, last), and
- is_writable_range(result, last - first, *first), and
- not_overlapped_forward(first, last, result)result + (last - first)

Ensures: Let i = copy(first, last, result) where

- i == result + (last - first), and
- equal(result, i, first).

Complexity: Exactly last - first copies.

The precondition of copy is that elements of the output range must not overlap the elements of the input range. The standard says simply that result must not be in the range [first, last), but this is very imprecise and leads to obvious questions. Namely, what does *in range* mean? Borrowing from EoP, we can construct a much more precise meaning of the requirement.

```
template<InputIterator I, WeaklyIncrementable O>
property not_overlapped_forward(I first1, I last1, O first2, O last2)
{
  is_readable_range(first1, last1) && is_readable_range(first2, last2) =>
    for all(I1 i : range(first1, last1))
      for all(I2 o : range(first2, last2))
        &*i == &*o => distance(first2, i) <= distance(first2, o);
}
```

This property guarantees that the algorithm does not read from an iterator after its value has been assigned. Note that we can only establish the property if the iter is a readable range. Furthermore, we can't write the Readable requirement as part of the template requirements because that would potentially make the property unsatisfiable for non-Readable output iterators. The precondition is vacuously true if you can't read from Out.

It may seem like a sensible thing to require that the output range must not contain cycles; that it is, it must be a counted range and not just a weak range. However, there are a number of use cases for cyclic, or seemingly cyclic output ranges, especially when there is no intent to read from the output range. We could, for example, think of an ostream_iterator as referring to a single element and each increment simply cycles to the same position: a kind of trivial state machine.

The postcondition can only be effectively evaluated for a subset of viable output iterators. In particular, the postcondition can only be verified when Out is a ForwardIterator. If Out is,

say, an ostream_iterator, we could not verify any of those properties without querying the state of the output stream.

```
template<WeakInputIterator I, WeaklyIncrementable Out>
requires IndirectlyCopyable<I, Out>
Out copy_n(I first, DistanceType<I> n, Out result);
```

Requires:

- is_readable_range(first, n), and
- is_writable_range(result, n, *first), and
- not_overlapped_forward(first, n, result).

Ensures: Let i = copy_n(first, n, result) where

- i == result + n, and
- equal(result, i, first).

Complexity: Exactly n copies.

The not_overlapped_forward requirement is an overload of the previous definition for weak ranges. It is defined as:

```
template<WeakInputIterator I, WeaklyIncrementable O>
axiom not_overlapped_forward(I first1, DistanceType<I> n, O first2)
{
  is_readable_range(first1, n) && is_readable_range(first2, n) =>
    for all(I i : range(first1, n))
      for all(O o : range(first2, n))
        &*i == &*o => distance(first1, i) <= distance(first2, o);
}
```

```
template<InputIterator I, WeaklyIncrementable Out, Semiregular P>
requires IndirectlyCopyable<I, Out> && Predicate<P, ValueType<I>>
Out copy_if(I first, I last, Out result, P pred);
```

Requires:

- is_readable_range(first, last)
- is_writable_range(first, count_if(first, last, pred), *first)
- not_overlapped_forward(first, last, result)result + count_if(first, last, p)

Ensures: Let i = copy_if(first, last, result, pred) where

- i == result + count_if(first, last, pred), and
- **for all**(I i : range(first, last) *i == *find_nth_if(first, last, pred).

Complexity: Exactly count_if(first, last, pred) copies.

The find_nth_if algorithm is an auxiliary function that returns the n^{th} iterator i in [first, last) that satisfies pred(*i).

```
template<BidirectionalIterator I, BidirectionalIterator Out>
requires IndirectlyCopyable<I, Out>
Out copy_backward(I first, I last, Out result);
```

Requires:

- is_readable_range(first, last), and
- is_writable_range(result - (last - first), result, *first), and
- not_overlapped_backward(first, last, result)result + (last - first).

Ensures: Let i = copy_backward(first, last, result) where

- i == result - (last - first), and
- equal(i, result, first) == true.

Complexity: Exactly last - first copies.

The not_overlapped_backward predicate defines the precondition for the copy_backward algorithm. It is similar to not_overlapped_forward:

```
template<InputIterator I, WeaklyIncrementable O>
property not_overlapped_backward(I first1, I last1, O first2, O last2)
{
  is_readable_range(first1, last1) && is_readable_range(first2, last2) =>
    for all(I i : range(first1, last1))
      for all(O o : range(first2, last2))
        &*i == &*o => distance(i, last1) <= distance(o, last2);
}
```

B.3.2 Move

```
template<InputIterator I, WeaklyIncrementable Out>
requires IndirectlyMovable<I, Out>
Out move(I first, I last,Out result);
```

Requires:

- is_readable_range(first, last), and
- is_movable_range(result, last - first, *first), and
- not_overlapped_forward(first, last, result)result + (last - first)

Ensures: Let i = move(first, last, result) where

- i == result + (last - first), and

- [result, i) has the values originally in [first, last), and
- the elements of [first, last) are partially formed.

Complexity: Exactly last - first moves.

```
template<BidirectionalIterator I, BidirectionalIterator Out>
requires IndirectlyMovable<I, Out>
Out move_backward(I first, I last, Out result);
```

Requires:

- is_readable_range(first, last), and
- is_writable_range(result - (last - first), result, *first), and
- not_overlapped_backward(first, last, result)result + (last - first).

Ensures: Let i = move_backward(first, last, result) where

- i == result - (last - first), and
- [i, result) has the values originally in [first, last), and
- The elements of [first, last) are partially formed.

Complexity: Exactly last - first copies.

B.3.3 Swap

```
template<InputIterator I1, WeakInputIterator I2>
requires IndirectlySwappable<I1, I2>
I2 swap_ranges(I1 first1, I1 last1, I2 first2);
```

Requires:

- is_permutable_range(first1, last1), and
- is_permutable_range(first2, last1 - first1)
- not_overlapped_forward(first1, last1, first2).

Ensures: Let last2 = swap_ranges(first1, last1, first2) where

- last2 - first2 == last1 - first1, and
- [first1, last1) has the values originally in [first2, last2), and
- [first2, last2) has the values originally in [first1, last1).

Complexity: Exactly last1 - first1 swaps.

```
template<Readable I1, Readable I2>
requires IndirectlySwappable<I1, I2>
void iter_swap(I1 i, I2 j);
```

Requires:

- is_valid(*i = move(*j))
- is_valid(*j = move(*i))

Ensures: The values pointed at by *i and *j are exchanged.

Complexity: Constant.

B.3.4 Transform

```
template<InputIterator I, WeaklyIncrementable Out, Function<ValueType<I> F>
requires Writable<ResultType<F, ValueType<I>>, Out>
Out transform(I first, I last, Out result, F f);
```

Requires:

- is_readable_range(first, last), and
- is_writable_range(result, last - first, f(*first))

Ensures: Let result_last = transform(first, last, result, f) where

- result_last == result + (last - first), and
- **for all**(DistanceType<I> i : range(0, last - first)) *(result + i) == f(*(first + i))

Complexity: f is applied exactly last - first times.

```
template<InputIterator I1,
         InputIterator I2,
         WeaklyIncrementable Out,
         Function<ValueType<I>, ValueType<I2>> F>
requires Writable<ResultType<F, ValueType<I1>, ValueType<I2>>, Out>
Out transform(I1 first1, I1 last1, I2 first2, Out result, F f);
```

Requires:

- is_readable_range(first1, last1), and
- is_readable_range(first2, last1 - first1), and
- is_writable_range(result, last1 - first1, f(*first1, *first2)).

Ensures: Let result_last = transform(first1, last1, first2, result, f) where

- result_last == result + (last1 - first1), and
- **for all**(DistanceType<I> i : range(0, last1 - first1)) *(result + i) == f(*(first1 + i), *(first2 + i))

Complexity: f is applied exactly last - first times.

B.3.5 Replace

```
template<InputIterator I, Semiregular T>
requires Writable<T, I> && EqualityComparable<ValueType<I>, T>
void replace(I first, I last, const T& old_value, const T& new_value);

template<InputIterator I, Semiregular P, Semiregular T>
requires Writable<T, I> && Predicate<P, ValueType<I>>
void replace_if(I first, I last, P pred, const T& new_value);
```

Requires: is_mutable_rangefirstlast.

Ensures: For the first overload, **for all**(DistanceType<I> i : range(0, last - first)) *(first + i) == (v[i] == old_value ? new_value : v[i]) where v = vector<ValueType<I>>{first, last} is copy of the original values in [first, last).

Ensures: For the second overload, **for all**(DistanceType<I> i : range(0, last - first)) *(first + i) == (pred(v[i]) ? new_value : v[i]) where v = vector<ValueType<I>>{first, last} is a copy of the original values in [first, last).

Complexity: Exactly last - first applications of operator == or the corresponding pred function and count(first, last, old_value) (or count_if(first, last, pred)) assignments.

```
template<InputIterator I, WeaklyIncrementable Out, Semiregular T>
requires IndirectlyCopyable<I, Out> && EqualityComparable<ValueType<I>, T> && Writable<T, Out>
Out replace_copy(I first, I last, Out result, const T& old_value, const T& new_value);

template<InputIterator I, WeaklyIncrementable Out, Semiregular P, Semiregular T>
requires IndirectlyCopyable<I, Out> && Predicate<P, ValueType<I>>&& Writable<T, Out>
Out replace_copy_if(I first, I last, Out result, P pred, const T& new_value);
```

Requires:

- is_readable_range(first, last), and
- is_writable_range(result, last - first, *first).

Ensures: For the first overload, let last_result = replace_copy(first, last, result, old_value, new_value) where

- last_result == result + (last - first), and
- **for all**(DistanceType<I> i : range(0, last - first)) *(result + i) == (*(first + i) == old_value ? new_value : *(first + i)).

Ensures: For the second overload, let last_result = replace_copy(first, last, result, pred, new_value) where

- last_result == result + (last - first), and
- **for all**(DistanceType<I> n : range(0, last - first)) *(result + i) == (pred(*(first + i), old_value) ? new_value : *(first + i)).

Complexity: Exactly last - first applications of == or the corresponding pred function and count(first, last, old_value) (or count_iffirstlastpred) assignments.

B.3.6 Fill

```
template<WeaklyIncrementable Out, Semiregular T>
requires EqualityComparable<Out> && Writable<T, Out>
void fill(Out first, Out last, const T& value);
```

Requires: is_writable_range(first, last, value).

Ensures: all_equal(first, last, value).

Complexity: Exactly last - first assignments.

```
template<WeaklyIncrementable Out, Semiregular T>
requires Writable<T, Out>
Out fill_n(Out first, DistanceType<Out> n, const T& value);
```

Requires: is_writable_range(first, n, value).

Ensures: Let last = fill_n(first, n, value) where

- last - first == n, and
- all_equal(first, n, value).

Complexity: Exactly n assignments.

B.3.7 Generate

```
template<WeaklyIncrementable Out, Function F>
requires EqualityComparable<Out> && Writable<ResultType<F>, Out>
F generate(Out first, Out last, F gen);
```

Requires: is_writable_range(first, last, gen()).

Ensures: generate(first, last, gen) is equivalent to:

```
if (first == last) {
  return move(gen);
} else {
  *first = gen();
  return move(generate(++first, last, gen));
}
```

Complexity: Exactly last - first applications of f.

```
template<WeaklyIncrementable Out, Function<> F>
requires Writable<ResultType<F>, Out>
std::pair<Out, F> generate_n(Out first, DistanceType<Out> n, F gen);
```

Requires: is_writable_range(first, n, gen()).

Ensures: generate(first, n, gen) is equivalent to:

```
if (n == 0) {
  return move(gen);
} else {
  *first = gen();
  return move(generate(++first, n, gen));
}
```

Complexity: Exactly n applications of f.

B.3.8 Remove

```
template<ForwardIterator I, EqualityComparable<ValueType<I>> T>
I remove(I first, I last, const T& value);

template<ForwardIterator I, Semiregular P>
requires Predicate<P, ValueType<I>> && Permutable<I>
I remove_if(I first, I last, P pred);
```

Requires: is_mutable_range(first, last).

Ensures: For the first overload, let i = remove(first, last, value) where find(first, i, value) == i.

Ensures: For the second overload, let i = remove_if(first, last, pred) where find_if(first, i, pred) == i.

Complexity: Exactly last - first applications of == or the corresponding pred function.

```
template<InputIterator I, WeaklyIncrementable Out, EqualityComparable T>
requires EqualityComparable<ValueType<I>, T> && IndirectlyCopyable<I, Out>
Out remove_copy(I first, I last, Out result, const T& value);

template<InputIterator I, WeaklyIncrementableOut, Semiregular P>
requires Predicate<P, ValueType<I>> && IndirectlyCopyable<I, Out>
Out remove_copy_if(I first, I last, Out result, P pred);
```

Requires:

- is_readable_range(first, last), and
- is_writable_range(result, last - first, *first).

Ensures: For the first overload, let last_result = remove_copy(first, last, result, value) where,

- last_result - result == count_not_equal(first, last, value), and
- findif(result, last_result, value) == last_result.

Ensures: For the second overload, let last_result = remove_copy_if(first, last, result, pred) where,

- last_result - result == count_if_not(first, last, pred), and
- find_if(result, last_result, pred) == last_result.

Complexity: Exactly last - first applications of == or the corresponding pred function and count_not_equal(first, last, value) (or count_if_not(first, last, pred) copies.

The count_not_equal and count_if_not algorithms are auxilliary functions that returns the number of elements that are not equal to value or do not satisfy pred. This is equivalent to the difference between last - first and the value returned by count_not_equal or count_if_not.

B.3.9 Unique

```
template<ForwardIterator I>
requires EqualityComparable<ValueType<I>> && Permutable<I>
I unique(I first, I last);

template<ForwardIterator I, Semiregular R>
requires Relation<R, ValueType<I>> && IndirectlyCopyable<I, I>
I unique(I first, I last, R comp);
```

Requires: is_mutable_range(first, last).

Ensures: For the first overload, let i = unique(first, last) where

- in_closed_rangeifirstlast, and
- adjacent_find(first, i) == i.

Ensures: For the second overload, let i = unique(first, last, comp) where

- in_closed_range(i, first, last), and
- adjacent_find(first, i, comp) == i.

Complexity: Exactly last - first - 1 applications of == or the corresponding comp relation.

```
template<InputIterator I, WeaklyIncrementable Out>
requires EqualityComparable<ValueType<I>> && IndirectlyCopyable<I, Out>
Out unique_copy(I first, I last, Out result);

template<InputIterator I, WeaklyIncrementable Out, Semiregular R>
requires Relation<R, ValueType<I>> && IndirectlyCopyable<I, Out>
Out unique_copy(I first, I last, Out result, R comp);
```

Requires:

- is_readable_range(first, last), and
- is_writable_range(result, last - first, *first).

Ensures: Let last_result = unique_copy(first, last, result) where

- is_bounded_rangeresultlast_result, and
- adjacent_find(result, last_result) == last_result.

Complexity: Exactly last - first - 1 applications of == or the corresponding comp relation.

B.3.10 Reverse

```
template<BidirectionalIterator I>
requires Permutable<I>
void reverse(I first, I last);
```

Requires: is_mutable_range(first, last).

Ensures: **for all**(DistanceType<I> i : range(0, last - first)) *(first + i) == v[v.size() - i - 1] where vector<ValueType<I>> v = {first, last} is a copy of the original values of [first, last).

Complexity: Exactly (last - first) / 2 swaps.

```
template<BidirectionalIterator I, WeaklyIncrementableOut>
requires IndirectlyCopyable<I, Out>
Out reverse_copy(I first, I last, Out result);
```

Requires:

- is_readable_range(first, last), and
- is_writable_range(result, last - first, *first).

Ensures: Let last_result = reverse_copy(first, last, result) where

- last_result == result + (last - first), and
- **for all**(DistanceType<I> i : range(0, last - first)) *(result + i) == *(first - i - 1)

Complexity: Exactly last - first copies.

B.3.11 Rotate

```
template<ForwardIterator I>
requires Permutable<I>
I rotate(I first, I middle, I last);
```

Requires:

- is_mutable_range(first, last), and
- in_closed_range(middle, first, last)

Ensures: Let vector<ValueType<I>> v = {first, last} be a copy of the original values in [first, last), and let i = rotate(first, last, middle) where

- i == first + (last - middle), and
- **for all**(DistanceType<I> n : range(0, last - first)) v[n] == *(first + (n + (last - middle)) % (last - first)).

Complexity: At most last - first swaps.

```
template<ForwardIterator I, WeaklyIncrementableOut>
requires IndirectlyCopyable<I, Out>
Out rotate_copy(I first, I middle, I last, Out result);
```

Requires:

- is_readable_range(first, last), and
- in_closed_range(middle, first, last)
- is_writable_range(result, last - first, *first), and

Ensures: Let last_result = rotate_copy(first, middle, last, result) where

- last_result == result + (last - first), and
- **for all**(DistanceType<I> n : range(0, last - first)) *(result + n) == *(first + (n + (last - middle)) % (last - first)).

Complexity: Exactly last - first copies.

B.3.12 Random Shuffle

```
template<RandomAccessIterator I>
requires Permutable<I>
void random_shuffle(I first, I last);

template<RandomAccessIterator I, Semiregular Gen>
requires Permutable<I> && RandomNumberGenerator<Gen, DistanceType<I>>
void random_shuffle(I first, I last, Gen&& rand);

template<RandomAccessIterator I, UniformRandomNumberGenerator Gen>
requires Permutable<I> && Convertible<ResultType<Gen>>
void shuffle(I first, I last, Gen&& g);
```

Requires: is_mutable_range(first, last).

Ensures: **for some**(auto&& r : permutations(first, last)) equal(begin(r), end(r), first).

Complexity: Exactly last - first - 1 swaps.

B.3.13 Partitions

```
template<InputIterator I, Semiregular P>
requires Predicate<P, ValueType<I>>
bool is_partitioned(I first, I last, P pred);
```

Requires: is_readable_range(first, last)

Ensures: Returns true if and only if

- first == last, or
- none_of(find_if_not(first, last, pred), last, pred).

Complexity: At most last - first applications of pred.

```
template<ForwardIterator I, Predicate<ValueType<I>> P>
requires Permutable<I>
I partition(I first, I last, P pred);

template<ForwardIterator I, Predicate<ValueType<I>> P>
requires Permutable<I>
I stable_partition(I first, I last, P pred);
```

Requires: is_permutable_range(first, last).

Ensures: Let i = partition(first, last, pred) where all_of(first, i, pred) is true and none_of(i, last, pred) is true. In the second overload, the partitioning is stable (the relative position of equal objects is preserved).

Complexity: For the first overload, at most last - first applications of pred.

Complexity: For the second overload, at most last - first * log(last - first) swaps and exactly last - first applications of pred. The number of swaps is linear if enough there is enough extra memory.

```
template<InputIterator I,
         WeaklyIncrementableOut1,
         WeaklyIncrementableOut2,
         Predicate<ValueType<I>> P>
requires IndirectlyCopyable<I, Out1> && IndirectlyCopyable<I, Out2>
pair<Out1, Out2> partition_copy(I first, I last, Out1 out_true, Out2 out_false, P pred);
```

Requires:

- is_readable_range(first, last), and
- is_writable_range(out_true, count_if(first, last, pred), *first), and
- is_writable_range(out_false, count_if_not(first, last, pred), *first).

Ensures: Let p = partition_copy(first, last, out_true, out_false, pred) where all_of(out_true, p.first, pred) is true and none_of(out_false, p.second, pred) is true.

Complexity: Exactly last - first copies and applications of pred.

```
template<ForwardIterator I, Semiregular P>
requires Predicate<P, ValueType<I>>
I partition_point(I first, I last, P pred);
```

Requires:

- is_readable_range(first, last), and
- is_partitioned(first, last, pred).

Ensures: Let i = partition_point(first, last, pred) where all_of(first, i, pred) is true and none_of(i, last, pred) is true.

Complexity: $\mathcal{O}$(log(last - first)) applications of pred.

B.4 Sorting and Related Algorithms

B.4.1 Sort

```
template<ForwardIterator I>
requires Sortable<I>
void sort(I first, I last);

template<ForwardIterator I>
requires Sortable<I>
void stable_sort(I first, I last);

template<ForwardIterator I, Semiregular R>
requires Sortable<I, R>
void sort(I first, I last, R comp);

template<ForwardIterator I, Semiregular R>
requires Sortable<I, R>
void stable_sort(I first, I last, R comp);
```

Requires: is_permutable_range(first, last). The third and fourth overloads also require strict_weak_ordering(comp).

Ensures: is_sorted(first, last) for the first two overlaods or is_sorted(first, last, comp) for the third and fourth. The stable_ algorithms also guarantee that the sorting is stable (the relative position of equal objects is preserved).

Complexity: For the unstable sort algorithms, $\mathcal{O}(n \log n)$ applications of < or the corresponding comp relation where n is last - first.

Complexity: For the stable_sort algorithms at most $n \log^2 n$ applications of < where n is last - first, $n \log n$ applications if there is sufficient memory.

```
template<RandomAccessIterator I>
requires Sortable<I>
void partial_sort(I first, I middle, I last);

template<RandomAccessIterator I, Semiregular R>
requires Sortable<I, R>
void partial_sort(I first, I middle, I last, R comp);
```

Requires:

- is_permutable_range(first, last), and
- in_range(first, last, middle).
- The second overload also requires strict_weak_ordering(comp).

Ensures: is_sorted(first, middle) or is_sorted(first, middle, comp) for the second overload.

Complexity: Approximately (last - first) * log(middle - first) applications of < or the corresponding comp relation.

```
template<InputIterator I1, RandomAccessIterator I2>
requires IndirectlyCopyable<I1, I2> && Sortable<I2>
I2 partial_sort_copy(I1 first, I1 last, I2 result_first, I2 result_last);

template<InputIterator I1, RandomAccessIterator I2, Semiregular R>
requires IndirectlyCopyable<I1, I2> && Sortable<I2, R>
I2 partial_sort_copy(I1 first, I1 last, I2 result_first, I2 result_last, R comp);
```

Requires:

- is_readable_range(first, last), and
- is_writable_range(result_first, result_last, *first).
- The second overload also requires strict_weak_ordering(comp).

Ensures: Let n = min(last - first, result_last - result_first) where is_sorted(result_first, result_first + n) (or is_sorted(result_first, result_first + n, comp) for the second overload).

Complexity: Approximately (last - first) * log(n) applications of < or the corresponding comp relation.

```
template<ForwardIterator I>
requires TotallyOrdered<ValueType<I>>
bool is_sorted(I first, I last);

template<ForwardIterator I, Relation<ValueType<I>>  R>
bool is_sorted(I first, I last, R comp);
```

Requires: is_readable_range(first, last)

Ensures: Returns true if and only if

- last - first < 2, or
- **for all**(DistanceType<I> i : range(0, last - first - 1))
  - for the first overload, !(*(first + i + 1) < *(first + i)).
  - for the second overload, !comp(*(first + i + 1), *(first + i)).

  Complexity: At most last - first applications of < or the corresponding comp relation.

```
template<ForwardIterator I>
requires TotallyOrdered<ValueType<I>>
I is_sorted_until(I first, I last);

template<ForwardIterator I, Relation<ValueType<I>> R>
I is_sorted_until(I first, I last, R comp);
```

Requires: is_readable_range(first, last). The second overload also requires strict_weak_ordering(comp).

Ensures: Let i = is_sorted_until(first, last) where is_sorted(first, i) (or is_sorted(first, i, comp) for the second overload) is true.

Complexity: At most last - first applications of < or the corresponding comp relation.

B.4.2 Nth Element

```
template<RandomAccessIterator I>
requires Sortable<I>
void nth_element(I first, I middle, I last);

template<RandomAccessIterator I, Semiregular R>
requires Sortable<I, R>
void nth_element(I first, I middle, I last, R comp);
```

Requires:

- is_mutable_range(first, last), and
- in_range(first, last, middle).
- The second overload also requires strict_weak_ordering(comp)

Ensures: Let vector<ValueType<I>> v = {first, last}, and sort(v.begin(), v.end()) (or sort(v.begin(), v.end(), comp) for the second overload) such that

- *middle == v[middle - first], and
- **for all**(I i : range(first, middle)) **for all**(I j : range(middle, last))
  - for the first overload, !(*i > *j)
  - for the second overload, !comp(*i, *j).

Complexity: Linear on average.

B.4.3 Binary Search

```
template<ForwardIterator I, TotallyOrdered T>
requires TotallyOrdered<VaueType<I>, T>
I lower_bound(I first, I last, const T& value);

template<ForwardIterator I, Relation<ValueType<I>> R>
I lower_bound(I first, I last, const ValueType<I>& value, R comp);
```

Requires:

- is_readable_range(first, last), and
- is_sorted(first, last) or is_sorted(first, last, comp).
- The second overload also requires strict_weak_ordering(comp).

Ensures: For the first overload, let i = lower_bound(first, last, value) where

- in_closed_range(i, first, last), and
- **for all**(I j : range(first, i)) *j < value, and
- *i != last => !(*i < value).

Ensures: For the second overload, let i = lower_bound(first, last, value, comp) where

- in_closed_range(i, first, last), and
- **for all**(I j : range(first, i)) comp(∗j, value), and
- i != last => !comp(∗i, value).

Complexity: At most log(last - first) +$\mathcal{O}(1)$ applications of < or the corresponding comp predicate.

```
template<ForwardIterator I, TotallyOrdered T>
requires TotallyOrdered<ValueType<I>, T>
I upper_bound(I first, I last, const T& value);

template<ForwardIterator I, Relation<ValueType<I>> R>
I upper_bound(I first, I last, const ValueType<I>& value, R comp);
```

Requires:

- is_readable_range(first, last), and
- is_sorted(first, last) or is_sorted(first, last, .)
- The second overload also requires strict_weak_ordering(comp).

Ensures: For the first overload, let i = upper_bound(first, last, value) where

- in_closed_range(i, first, last), and
- **for all**(I j : range(first, i)) !(value < ∗j), and
- i != last => ∗i < value.

Ensures: For the second overload, let i = upper_bound(first, last, value, comp) where

- in_closed_range(i, first, last), and
- **for all**(I j : range(first, i)) !comp(value, ∗j), and
- i != last => comp(∗i, value).

Complexity: At most log(last - first) +$\mathcal{O}(1)$ applications of < or the corresponding comp predicate.

```
template<ForwardIterator I, TotallyOrdered T>
requires TotallyOrdered<ValueType<I>, T>
pair<I, I> equal_range(I first, I last, const T& value);

template<ForwardIterator I, Relation<ValueType<I>> R>
pair<I, I> equal_range(I first, I last, const ValueType<I>& value, R comp);
```

Requires:

- is_readable_range(first, last), and
- is_sorted(first, last) or is_sorted(first, last, comp).
- The second overload also requires strict_weak_ordering(comp).

Ensures: For the first overload, let p = equal_range(first, last, value) where

- p.first == lower_bound(first, last, value), and
- p.second == upper_bound(first, last, value), and
- all_equal(p.first, p.second, value).

Ensures: For the second overload, let p = equal_range(first, last, value, comp) where

- p.first == lower_bound(first, last, value, comp), and
- p.second == upper_bound(first, last, value, comp), and
- all_if(p.first, p.second, symmetric_complement(comp)).

Complexity: At most 2 * log(last - first) +$\mathcal{O}(1)$ applications of < or the corresponding comp predicate.

```
template<ForwardIterator I, TotallyOrdered<ValueType<I>> T>
bool binary_search(I first, I last, const T& value);

template<ForwardIterator I, Semiregular R>
requires Relation<R, ValueType<I>>
bool binary_search(I first, I last, const ValueType<I>& value, R comp);
```

Requires:

- is_readable_range(first, last), and
- is_sorted(first, last) or is_sorted(first, last, comp).
- The second overload also requires strict_weak_ordering(comp).

Ensures: Returns true if and only if, when p = equal_range(first, last, value), p.first != p.second (or p = equal_range(first, last, comp) for the second overload).

Complexity: At most log(last - first) +$\mathcal{O}(1)$ applications of < or the corresponding comp predicate.

B.4.4 Merge

```
template<InputIterator I1, InputIterator I2, WeaklyIncrementableOut>
requires Mergeable<I1, I2, Out>
Out merge(I1 first1, I1 last1, I2 first2, I2 last2, Out result);

template<InputIterator I1,
         InputIterator I2,
```

```
        WeaklyIncrementable Out,
        Relation<ValueType<I1>, ValueType<I2> R>
requires Mergeable<I1, I2, Out, R>
Out merge(I1 first1, I1 last1, I2 first2, I2 last2, Out result, R comp);
```

Requires: can_merge_ranges(first1, last1, first2, last2, result) or can_merge_ranges(first1, last1, first2, last2, result, comp) for the second overload. The second overload also requires strict_weak_ordering(comp).

Ensures: For the first overload, let result_last = merge(first1, last1, first2, last2, out) where

- includes(result, result_last, first1, last1) == true, and
- includes(result, result_last, first2, last2) == true.

Ensures: For the second overload, let result_last = merge(first1, last1, first2, last2, out, comp) where

- includes(result, result_last, first1, last1, comp) == true, and
- includes(result, result_last, first2, last2, comp) == true.

Complexity: At most (last1 - first1) + (last2 - first2) - 1 applications of < or the corresponding comp relation.

The can_merge_ranges predicate is defined as:

```
template<InputIterator I1, InputIterator I2, WeaklyIncrementableOut>
requires Mergeable<I1, I2, Out>
property can_merge_ranges(I1 first1, I1 last1, I2 first2, I2 last2, Out result)
{
  is_readable_range(first1, last1) && is_sorted(first1, last1) &&
  is_readable_range(first2, last2) && is_sorted(first2, last2) &&
  not_overlapped(first1, last1, first2) &&
  is_writable_range(result, (last1 - first1) + (last2 - first2), *first1) &&;
  is_writable_range(result, (last1 - first1) + (last2 - first2), *first2);
}
```

A corresponding overload for relations can be easily derived.

```
template<ForwardIterator I>
requires Sortable<I>
void inplace_merge(I first, I middle, I last);

template<ForwardIterator I, Semiregular R>
requires Sortable<I, R>
void inplace_merge(I first, I middle, I last, R comp);
```

Requires:

- is_permutable_range(first, last), and
- in_range(first, last, middle), and
- is_sorted(first, middle), and
- is_sorted(middle, last).

- The second overload also requires strict_weak_ordering(comp).

Ensures: For the first overload, inplace_merge(first, middle, last) is equivalent to the following program:

```
vector<ValueType<I>> v(last - first);
auto v_last = merge(first, middle, middle, last, v.begin());
copy(v.begin(), v_last, first);
```

Ensures: For the second overload, inplace_merge(first, middle, last, comp) is equivalent to the following program:

```
vector<ValueType<I>> v(last - first);
merge(first, middle, middle, last, v.begin(), comp);
copy(v.begin(), v.end(), first);
```

Complexity: With enough memory available, at most last - first applications of < or the corresponding comp relation. Otherwise $\mathcal{O}(n \log n)$ applications where n is last - first.

B.4.5 Set Operations

```
template<InputIterator I1, InputIterator I2>
requires TotallyOrdered<ValueType<I1>, ValueType<I2>>
bool includes(I1 first1, I1 last1, I2 first2, I2 last2);

template<InputIterator I1, InputIterator I2, Relation<ValueType<I1>, ValueType<I2>> R>
bool includes(I1 first1, I1 last1, I2 first2, I2 last2, R comp);
```

Requires:

- is_sorted(first1, last1), and
- is_sorted(first1, last1).
- The second overload also requires strict_weak_ordering(comp).

Ensures: Returns true if and only if every element in [first2, last2) can be found in [first1, last1).

Complexity: At most 2 * ((last1 - first1) + (last2 - first2)) - 1 applications of < or the corresponding comp relation.

```
template<InputIterator I1, InputIterator I2, WeaklyIncrementable Out>
requires Mergeable<I1, I2, Out>
Out set_union(I1 first1, I1 last1, I2 first2, I2 last2, Out result);

template<InputIterator I1,
         InputIterator I2,
         WeaklyIncrementable Out,
         Relation<ValueType<I1>, ValueType<I2>> R>
requires Mergeable<I1, I2, Out, R>
Out set_union(I1 first1, I1 last1, I2 first2, I2 last2, Out result, R comp);
```

Requires: can_merge_ranges(first1, last1, first2, last2, result) or can_merge_ranges(first1, last1, first2, last2, result, comp) for the second overload.

Ensures: For the first overload, let result_last = set_union(first1, last1, first2, last2, result) where

- is_sorted(result, result_last), and
- **for all**(Out i : range(result, result_last)) find(first1, last1, *i) != last1 || find(first2, last2, *i) != last.

Ensures: For the second overload, let result_last = set_union(first1, last1, first2, last2, result, comp) where

- is_sorted(result, result_last, comp), and
- **for all**(Out i : range(result, result_last)) find(first1, last1, *i) != last1 || find(first2, last2, *i) != last.

Complexity: At most 2 * ((last1 - first1) + (last2 - first2)) - 1 applications of < or the corresponding pred comp.

For both overloads, if [first1, last1) contains m equal elements and [first2, last2) contains n elements equal to those in the first range, then all m elements from the first range are copied into the output range (in order) and max(n - m, 0) elements from the second range are copied (in order).

```
template<InputIterator I1, InputIterator I2, WeaklyIncrementable Out>
requires Mergeable<I1, I2, Out>
Out set_intersection(I1 first1, I1 last1, I2 first2, I2 last2, Out result);

template<InputIterator I1,
         InputIterator I2,
         WeaklyIncrementable Out,
         Relation<ValueType<I1>, ValueType<I2>> R>
requires Mergeable<I1, I2, Out, R>
Out set_intersection(I1 first1, I1 last1, I2 first2, I2 last2, Out result, R comp);
```

Requires: can_merge_ranges(first1, last1, first2, last2, result) or can_merge_ranges(first1, last1, first2, last2, result, comp) for the second overload.

Ensures: For the first overload, let result_last = set_intersection(first1, last1, first2, last2, result) where

- is_sorted(result, result_last), and
- **for all**(Out i : range(result, result_last)) find(first1, last1, *i) != last1 && find(first2, last2, *i) != last.

Ensures: For the second overload, let result_last = set_intersection(first1, last1, first2, last2, result, comp) where

- is_sorted(result, result_last, comp), and
- **for all**(Out i : range(result, result_last)) find(first1, last1, *i) != last1 && find(first2, last2, *i) != last.

Complexity: At most 2 * ((last1 - first1) + (last2 - first2)) - 1 applications of < or the corresponding comp relation.

For both overloads, if [first1, last1) contains m elements that are equivalent to each other and [first2, last2) contains n elements that are equivalent to those in the first range, only the first min(m, n) elements are copied from the first range to the output range (in order).

```
template<InputIterator I1, InputIterator I2, WeaklyIncrementable Out>
requires Mergeable<I1, I2, Out>
Out set_difference(I1 first1, I1 last1, I2 first2, I2 last2, Out result);

template<InputIterator I1,
         InputIterator I2,
         WeaklyIncrementable Out,
         Relation<ValueType<I1>, ValueType<I2>> R>
requires Mergeable<I1, I2, Out, R>
Out set_difference(I1 first1, I1 last1, I2 first2, I2 last2, Out result, R comp);
```

Requires: can_merge_ranges(first1, last1, first2, last2, result) or can_merge_ranges(first1, last1, first2, last2, result, comp) for the second overload.

Ensures: For the first overload, let result_last = set_difference(first1, last1, first2, last2, result) where

- is_sorted(result, result_last), and
- **for all**(Out i : range(result, result_last)) find(first1, last1, *i) != last1 && find(first2, last2, *i) == last.

Ensures: For the second overload, let result_last = set_difference(first1, last1, first2, last2, result, comp) where

- is_sorted(result, result_last, comp), and
- **for all**(Out i : range(result, result_last)) find(first1, last1, *i) != last1 && find(first2, last2, *i) == last.

Complexity: At most 2 * ((last1 - first1) + (last2 - first2)) - 1 applications of < or the corresponding comp relation.

For both overloads, if [first1, last1) contains m elements that are equivalent to each other and [first2, last2) contains n elements that are equivalent to those in the first range, the last max(m - n, 0) elements from the first range are copied to the output range.

```
template<InputIterator I1, InputIterator I2, WeaklyIncrementable Out>
requires Mergeable<I1, I2, Out>
Out set_symmetric_difference(I1 first1, I1 last1, I2 first2, I2 last2, Out result);

template<InputIterator I1,
         InputIterator I2,
         WeaklyIncrementable Out,
         Relation<ValueType<I1>, ValueType<I2>> R>
Out set_symmetric_difference(I1 first1, I1 last1, I2 first2, I2 last2, Out result, R comp);
```

Requires: can_merge_ranges(first1, last1, first2, last2, result) or can_merge_ranges(first1, last1, first2, last2, result, comp) for the second overload.

Ensures: Let result_last = set_symmetric_difference(first1, last1, first2, last2, result) where

- is_sorted(result, result_last), and
- **for all**(Out i : range(result, result_last)) find(first1, last1, *i) != last1 <=> find(first2, last2, *i) == last.

Ensures: Let result_last = set_symmetric_difference(first1, last1, first2, last2, result, comp) where

- is_sorted(result, result_last, comp), and
- **for all**(Out i : range(result, result_last)) find(first1, last1, *i) != last1 <=> find(first2, last2, *i) == last.

Complexity: At most 2 * ((last1 - first1) + (last2 - first2)) - 1 applications of < or the corresponding comp relation.

For both overloads, if [first1, last1) contains m elements that are equivalent to each other and [first2, last2) contains n elements that are equivalent to those in the first range, then abs(m - n) of those elements shall be copied to the output range. The elements copied are the last m - n of these elements from [first1, last1) if m > n, and the last n - m of these elements from [first2, last2) if m < n.

B.4.6 Heap Operations

```
template<RandomAccessIterator I>
requires Sortable<I>
void push_heap(I first, I last);

template<RandomAccessIterator I, Relation<ValueType<I>> R>
requires Sortable<I, R>
void push_heap(I first, I last, R comp);
```

Requires:

- first != last, and
- is_permutable_range(first, last), and
- is_heap(first, last - 1) or is_heap(first, last - 1, comp) for the second overload.

Ensures: is_heap(first, last) is true or, for the second overload, is_heap(first, last, comp) is true.

Complexity: At most log(last - first) applications of < or the corresponding comp relation.

```
template<RandomAccessIterator I>
requires Sortable<I>
void pop_heap(I first, I last);

template<RandomAccessIterator I, Relation<ValueType<I>> R>
requires Sortable<I, R>
void pop_heap(I first, I last, R comp);
```

Requires:

- first != last, and

- is_permutable_range(first, last), and
- is_heap(first, last) or is_heap(first, last, comp) for the second overload.

Ensures: Let x = *first be the original value referenced by first such that, after calling pop_heap(first, last) (or pop_heap(first, last, comp)).

- x == *(last - 1), and
- is_heap(first, last - 1) is true or is_heap(first, last - 1, comp) for the second overload.

Complexity: At most 2 * log(last - first) applications of < or the corresponding comp relation.

```
template<RandomAccessIterator I>
requires Sortable<I>
void make_heap(I first, I last);

template<RandomAccessIterator I, Relation<ValueType<I>> R>
requires Sortable<I, R>
void make_heap(I first, I last, R comp);
```

Requires: is_permutable_range(first, last). The second overload also requires strict_weak_ordering(comp).

Ensures: is_heap(first, last) is true or is_heap(first, last, comp) for the second overload.

Complexity: At most 3 * last - first applications of < or the corresponding comp relation.

```
template<RandomAccessIterator I>
requires Sortable<I>
void sort_heap(I first, I last);

template<RandomAccessIterator I, Relation<ValueType<I>> R>
requires Sortable<I, R>
void sort_heap(I first, I last, R comp);
```

Requires:

- is_permutable_range(first, last), and
- is_heap(first, last) or is_heap(first, last, comp).
- The second overload also requires strict_weak_ordering(comp).

Ensures: is_sorted(first, last) is true or is_sorted(first, last, comp) for the second overload.

Complexity: At most $\mathcal{O}(n \log n)$ applications < or the corresponding comp relation where n is last - first.

```
template<ForwardIterator I>
requires TotallyOrdered<ValueType<I>>
bool is_heap(I first, I last);

template<ForwardIterator I,  Relation<ValueType<I>> R>
bool is_heap(I first, I last, R comp);
```

Requires: is_readable_range(first, last). The second overload also requires strict_weak_ordering(comp).

Ensures: **for all**(DistanceType<I> i : range(0, last - first)) is_heap_ordered(first, last, i). For the second overload, the corresponding predicate is is_heap_ordered(first, last, i, comp).

Complexity: Linear.

The is_heap_ordered evaluates the relation between a node in the heap and its children. Because the C++ standard does not specify the arity of the heap, we cannot write a "naked" formula to define the relationship. The predicate has the following signatures:

```
template<ForwardIterator I>
requires TotallyOrdered<ValueType<I>>
bool is_heap_ordered(I first, I last, DistanceType<I> n);

template<ForwardIterator I,  Relation<ValueType<I>> R>
bool is_heap_ordered(I first, I last, DistanceType<I> n, R comp);
```

The operation returns true if and only if the ith element of [first, last) is not less than its children (if any). That is, for all children c of element i, it must be that !(*(first + i) < *(first + c)). The number of children and their offsets depend on the implementation of the heap (binary, ternary, etc.). The second overload replaces the < with a corresponding strict weak order.

```
template<ForwardIterator I>
requires TotallyOrdered<ValueType<I>>
I is_heap_until(I first, I last);

template<ForwardIterator I, Semiregular R>
requires Relation<R, ValueType<I>>
I is_heap_until(I first, I last, R comp);
```

Requires: is_readable_range(first, last). The second overload also requires strict_weak_ordering(comp).

Ensures: For the first overload, let i = is_heap_until(first, last) where

- in_closed_range(i, first, last), and
- **for all**(I j : range(first, i)) is_heap(first, j), and
- i != last => !is_heap(first, i + 1).

Ensures: For the second overload, let i = is_heap_until(first, last, comp) where

- in_closed_range(i, first, last), and
- **for all**(I j : range(first, i)) is_heap(first, j, comp), and
- i != last => !is_heap(first, i + 1, comp).

Complexity: Linear.

B.4.7 Minimum and Maximum

```
template<TotallyOrdered T>
const T& min(const T& a, const T& b);

template<Semiregular T, Semiregular R>
requires Relation<R, T>
const T& min(const T& a, const T& b, R comp);
```

Requires: The second overload requires strict_weak_ordering(comp)a.

Ensures: For the first overload, let x = min(a, b)) where x <= a && x <= b.

Ensures: For the second overload, let x = min(a, b, comp) where !comp(a, x) && !comp(b, x).

Complexity: Exactly one application of < of comp.

```
template<Semiregular T>
requires TotallyOrdered<T>
const T& min(initializer_list<T> t);

template<Semiregular T, Relation<T> R>
const T& min(initializer_list<T> t, R comp);
```

Requires: The second overload requires strict_weak_ordering(comp).

Ensures: Equivalent to *min_element(t.begin(), t.end()) or *min_element(t.begin(), t.end()) for the second overload.

Complexity: Exactly t.size() - 1 applications of < or the corresponding comp relation.

```
template<TotallyOrdered T>
const T& max(const T& a, const T& b);

template<Semiregular T, Relation<T> R>
const T& max(const T& a, const T& b, R comp);
```

Requires: The second overload requires strict_weak_ordering(comp).

Ensures: For the first overload, x = min(a, b)) where x >= a && x >= b.

Ensures: For the second overload, let x = min(a, b)) such that !comp(x, a) && !comp(x, b).

Complexity: Exactly one application of <.

```
template<Semiregular T>
requires TotallyOrdered<T>
T max(initializer_list<T> t);

template<Semiregular T, Relation<T> R>
T max(initializer_list<T> t, R comp);
```

Requires: The second overload requires strict_weak_ordering(comp).

Ensures: Equivalent to *max_elemnt(t.begin(), t.end()) or *max_element(t.begin(), t.end()) for the second overload.

Complexity: Exactly t.size() - 1 applications of < or the corresponding comp relation.

```
template<TotallyOrdered T>
pair<const T&, const T&> minmax(const T& a, const T& b);

template<Semiregular T, Relation<T> R>
pair<const T&, const T&> minmax(const T& a, const T& b, R comp);
```

Requires: The second overload requires strict_weak_ordering(comp).

Ensures: For the first overload, let p = minmax(a, b) where p.first == min(a, b) && p.second == max(a, b).

Ensures: For the second overload, let p = minmax(a, b, comp) where p.first == min(a, b, comp) && p.second == max(a, b, comp).

Complexity: Exactly one application of < or comp.

```
template<TotallyOrdered T>
pair<const T&, const T&> minmax(initializer_list<T> t);

template<Semiregular T, Relation<T> R>
pair<const T&, const T&> minmax(initializer_list<T> t, R comp);
```

Requires: The second overload requires strict_weak_ordering(comp).

Ensures: For the first overload, let p = minmax(t) where p.first == min(t) and p.second == max(t).

Ensures: For the second overload, let p = minmax(t, comp) where p.first == min(t, comp) and p.second == max(t, comp).

Complexity: Exactly t.size() - 1 applications of comp or the corresponding comp relation.

```
template<ForwardIterator I>
requires TotallyOrdered<ValueType<I>>
I min_element(I first, I last);

template<ForwardIterator I, Relation<ValueType<I>> R>
I min_element(I first, I last, R comp);
```

Requires: is_readable_range(first, last). The second overload also requries strict_weak_orderingcomp.

Ensures: For the first overload, i = min_element(first, last) where

- in_range(i, first, last), and
- **for all**(I j : range(first, last)) min(*i, *j) == *i.

Ensures: For the second overload, i = min_element(first, last, comp) where

- in_range(i, first, last), and
- **for all**(I j : range(first, last)) min(*i, *j, comp) == *i.

Complexity: Exactly max((last - first) - 1, 0) applications of operator < or the corresponding comp relation.

```
template<ForwardIterator I>
requires TotallyOrdered<ValueType<I>>
I max_element(I first, I last);

template<ForwardIterator I, Relation<ValueType<I>> R>
I max_element(I first, I last, R comp);
```

Requires: is_readable_range(first, last). The second overload also requries strict_weak_orderingcomp.

Ensures: For the first overload, i = max_element(first, last) where

- in_range(i, first, last), and
- **for all**(I j : range(first, last)) max(*i, *j) == *i.

Ensures: For the second overload, i = max_element(first, last, comp) where

- in_range(i, first, last), and
- **for all**(I j : range(first, last)) max(*i, *j, comp) == *i.

Complexity: Exactly max((last - first) - 1, 0) applications of operator < or the corresponding comp relation.

```
template<ForwardIterator I>
requires TotallyOrdered<ValueType<I>>
pair<I, I> minmax_element(I first, I last);

template<ForwardIterator I, Relation<ValueType<I>> R>
pair<I, I> minmax_element(I first, I last, R comp);
```

Requires: is_readable_range(first, last). The second overload also requries strict_weak_orderingcomp.

Ensures: For the first overload, p = minmax_element(first, last) where

- in_range(p.first, first, last), and
- in_range(p.second, first, last), and
- p.first == min_element(first, last) && p.second == max_element(first, last).

Ensures: For the first overload, p = minmax_element(first, last, comp) where

- in_range(p.first, first, last), and
- in_range(p.second, first, last), and p.first == min_element(first, last, comp) && p.second == max_element(first, last, comp).

Complexity: Exactly max((last - first) - 1, 0) applications of operator < or the corresponding comp relation.

B.4.8 Lexicographical Comparison

```
template<InputIterator I1, InputIterator I2>
requires TotallyOrdered<ValueType<I1>, ValueType<I2>>
bool lexicographical_compare(I1 first1, I1 last1, I2 first2, I2 last2);

template<InputIterator I1, InputIterator I2, Relation<ValueType<I1>, ValueType<I2>> R>
bool lexicographical_compare(I1 first1, I1 last1, I2 first2, I2 last2, R comp);
```

Requires:

- is_readable_range(first1, last1), and
- is_readable_range(first2, last2), and
- the second overload also requires strict_weak_ordering(comp).

Ensures: The first overload returns true if and only if

- **for all**(DistanceType<I1> i : range(0, min(last1 - first1, last2 - first2))) *(first1 + i) < *(first2 + i), and
- (last1 - first1) < (last2 -first2).

Ensures: The second overload returns true if and only if

- **for all**(DistanceType<I1> i : range(0, min(last1 - first1, last2 - first2))) comp(*(first1 + i), *(first2 + i)), and
- (last1 - first1) < (last2 -first2).

Complexity: At most 2 * min(last1 - first1, last2 - first2) applications of < or the corresponding comp relation.

B.4.9 Permutations

```
template<BidirectionalIterator I>
requires Sortable<I>
bool next_permutation(I first, I last);

template<BidirectionalIterator I, Relation<ValueType<I>> R>
requires Sortable<I, R>
bool next_permutation(I first, I last, R comp);
```

Requires:

- is_permutable_range(first, last)
- The second overload also requires strict_weak_ordering(comp).

Ensures: For the first overload, let vector<ValueType<I>> v{first, last} and b = next_permutation(first, last). If b is true, then is_permutation(first, last, v.begin()) is true and lexicographical_compare(v.begin(), v.end(), first, i)s true. If b is false, then is_sorted(first, last) is true.

Ensures: For the second overload, let vector<ValueType<I>> v{first, last} and b = next_permutation(first, last, comp). If b is true, then is_permutation(first, last, v.begin(), symmetric_complement<R>(comp))

is true and lexicographical_compare(v.begin(), v.end(), first, last, comp) is true. If b is false, then is_sorted(first, last, comp) is true.

Complexity: At most (last - first) / 2 swaps.

The symmetric_complement function object used in the specification of the second overload, evaluates the expression !comp(a, b) && !comp(b, a). It determines if the two sequences are permutations with the respect to the equivalence relation defined by the strict weak ordering comp.

```
template<BidirectionalIterator I>
requires Sortable<I>
bool prev_permutation(I first, I last);

template<BidirectionalIterator I, Semiregular R>
requires Sortable<I, R>
bool prev_permutation(I first, I last, R comp)
```

Requires:

- is_permutable_range(first, last)
- The second overload also requires strict_weak_ordering(comp).

Ensures: For the first overload, let vector<ValueType<I>> v{first, last} and b = next_permutation(first, last). If b is true, then is_permutation(first, last, v.begin()) is true and lexicographical_compare(first, last, v.begin(), i)s true. If b is false, then is_sorted(first, last) is true.

Ensures: For the second overload, let vector<ValueType<I>> v{first, last} and b = next_permutation(first, last, comp). If b is true, then is_permutation(first, last, v.begin(), symmetric_complement<R>(comp)) is true and lexicographical_compare(first, last, v.begin(), v.end(), comp) is true. If b is false, then is_sorted(first, last, comp) is true.

Complexity: At most (last - first) / 2 swaps.

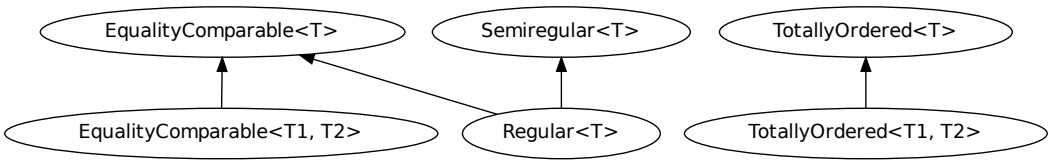
Appendix C Concept Summary

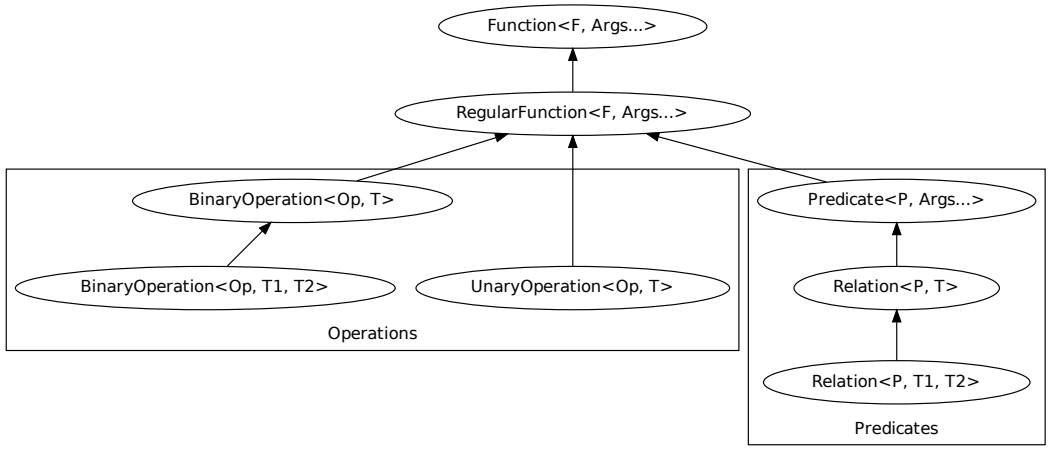
This appendix includes diagrams relating the concept structure and an index relating each concept to its uses.

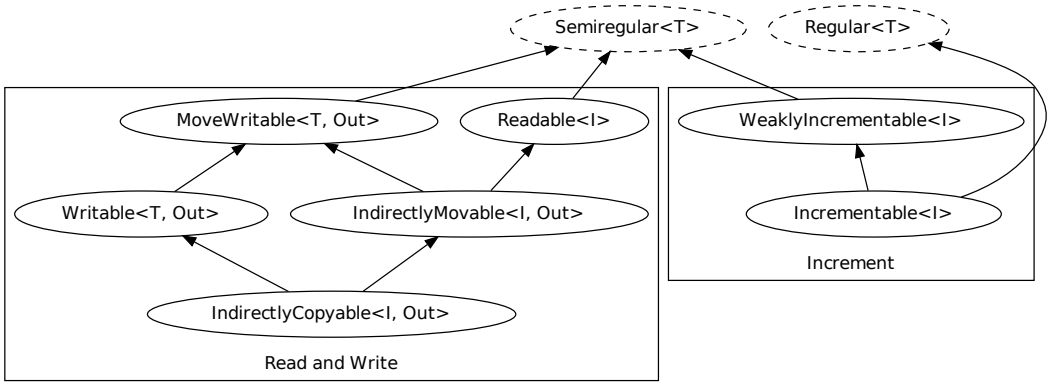
C.1 Concept Diagrams

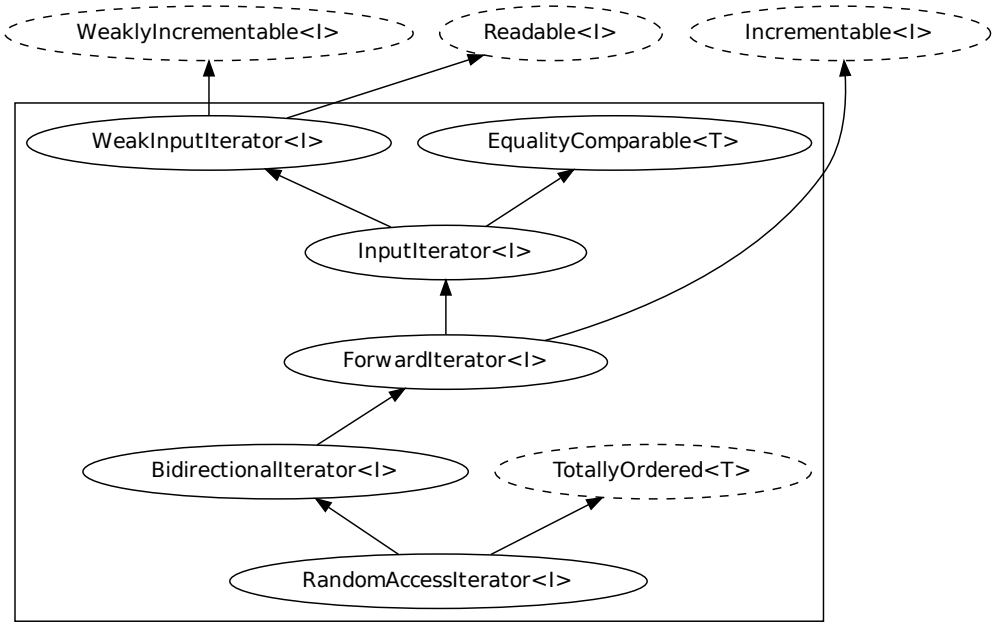
The following diagrams show the relationships between the concepts described in the technical report. The edges between concepts do not indicate the *exact* requirements, but rather subsets of individual syntactic requirements. The intent is to capture the relationship between abstractions rather than the actual requirements. For example, we could draw Incrementable as having arrows to both WeaklyIncrementable and EqualityComparable. Instead we opt to draw connections between WeaklyIncrementable and Regular. If a type is Incrementable, then it is guaranteed to be both WeaklyIncrementable and Regular.

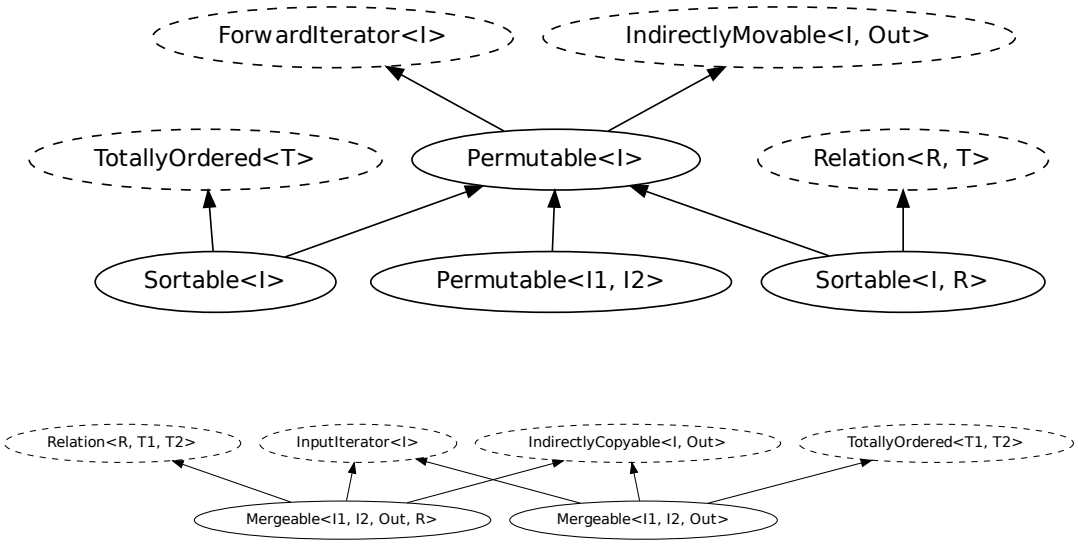
These diagrams omit edges that represent requirements to related type. For example WeakInputIterator requires its ValueType to be Integral; that relationship is not shown.






Semiregular<T>
Regular<T>
MoveWritable<T, Out>
Readable<I>
WeaklyIncrementable<I>
Writable<T, Out>
IndirectlyMovable<I, Out>
Incrementable<I>
Increment
IndirectlyCopyable<I, Out>
Read and Write


WeaklyIncrementable<I>
Readable<I>
Incrementable<I>
WeakInputIterator<I>
EqualityComparable<T>
InputIterator<I>
ForwardIterator<I>
BidirectionalIterator<I>
TotallyOrdered<T>
RandomAccessIterator<I>



C.2 Concept Cross-reference

The concept index relates each concept to its uses in the technical report. Each concept listed in the appendix is used by each of the algorithms and concepts underneath it.

Appendix D Alternative Designs

The design presented in this report prioritizes the representation of general ideas over exact requirements (§1.3). We think that concepts in this report represent sound descriptions of the basic ideas in the STL and serve as a good starting point for the discussion of requirements for a standard library. At the same time, we recognize that our design is not wholly compatible with the existing C++11 standard, and it constrains algorithms more than strictly necessary. The exact balance between the clarity of design and the exactness of requirements has to be the subject of a careful discussion that we hope this report will inspire.

D.1 Decomposing Semiregular

In this section we give an alternative design that decomposes the Semiregular concept into Movable and Copyable requirements. The Semiregular concept is required in a few places where temporaries are expected to be created. This has the effect of inducing copy and default construction requirements where they are not strictly required by the algorithm. This decomposition allows us to use the value-oriented STL algorithms with non-regular types such as non-copyable resources (e.g. using reverse with a range of unique_ptrs).

The Movable concept describes *move semantics*: types that can be both move constructed and assigned.

```
concept Movable<typename T> =
  requires object (T a) {
    T* == {&a};
    axiom { &a == addressof(a); }

    a.~T();
    noexcept(a.~T());
  } &&
  requires initialization (T a, T b, T c) {
    // Move construction
    T{move(a)};
    axiom { eq(a, b) => eq(T{move(a)}, b); }

    // Move assignment
    T& == {a = move(b)};
    axiom { eq(b, c) => eq(a = move(b), c); }

    // Allocation
    T* == new T{move(a)};
    delete new T{move(a)};
  };
```

In addition to expressing the requirements of move semantics, the Movable concept also captures the basic object requirements, which include the ability to take an object's address and destruction. The axioms of move formalize the standard requirement (C++ Standard, utility.arg.requirements (tables 20, 22)) that the result of move construction or assignment produces results in an object that is equal to the source object prior to the move.

The reason that we require both construction and assignment is that regular types can be used to construct variables. That means that we can both initialize an object and assign values to it. We do not expect the pairing of these requirements to have any significant impact on existing or future programs since these operations are frequently provided together.

The Copyable concept represents the analogous copy semantics: types that can be copy constructed and copy assigned.

```
concept Copyable<Movable T> =
  requires initialization (T a, T b, T c) {
    // Copy construction
    T{a};
    axiom { eq(T{a}, a); }

    // Copy assignment
    T& == {a = b};
    axiom { eq(a = b, b); }
  } &&
  requires allocation () {
    T* == { new T{a}; };
    delete new T{a};
  };
};
```

The Copyable concept "inherits" a number of requirements from the Movable concept. In essence, a Copyable type is a Semiregular type that may not be default constructible. The allocation requirements have been adapted accordingly; the concept does not permit the allocation of an array of T objects.

It is not strictly necessary for the Copyable concept to require Movable. We could have written the concepts to be completely independent. However, that leads to restatements of many basic requirements (e.g. destruction).

Given these new concepts, we can simply define Semiregular as being Copyable and default constructible:

```
concept Semiregular<Copyable T> =
  requires DefaultConstructible {
    T{};
  } &&
  requires Allocatable() {
    T* == { new T };
    delete[] new T[1];
  };
```

The definition is equivalent to how Semiregular is defined in our design, but the two additional concepts allow more fine-grained specifications of requirements for algorithms.

The only other concept affected by the design change is Permutable. Its new definition is:

```
concept Permutable<ForwardIterator I> =
  Movable<ValueType<I>> &&
  IndirectlyMovable<I, I>;
```

We have simply replaced the previous Semiregular concept with Movable. This change impacts every algorithm that requires Permutable, including all of the Sortable algorithms.

The new design requires changes to three algorithms: iter_swap, swap_Ranges, and sort. As with the Permutable concept, we replace the Semiregular requirement with Movable for iter_swap and swap_Ranges. Their definitions become:

```
template<Readable I1, Readable I2>
requires IndirectlyMovable<I2, I1> && IndirectlyMovable<I1, I2>
    && Movable<ValueType<I1>> && MovableValueType<I2>>
void iter_swap(I1 i, I2 j);
```

```
template<InputIterator I1, WeakInputIterator I2>
requires IndirectlyMovable<I2, I1> && IndirectlyMovable<I1, I2>
    && Movable<ValueType<I1>> && MovableValueType<I2>>
I2 swap_ranges(I1 first1, I1 last1, I2 first2);
```

For the sort algorithm, we modify the signature to include a Regular requirement on the iterator argument's ValueType.

```
template<ForwardIterator I>
requires Regular<ValueType<I>> && Sortable<I>
void sort(I first, I last);

template<ForwardIterator I, Relation<ValueType<I>> R>
requires Regular<ValueType<I>> && Sortable<I, R>
void sort(I first, I last, R comp);
```

Recall that the quicksort algorithm requires us to make a copy of the pivot element (Hoare, 1969). We require Regular instead of Copyable because the ValueType is already required to be Movable (through Permutable) and EqualityComparable (through TotallyOrdered). We think that stating the strongest requirement communicates the intent more effectively than stating the least redundant requirement.

Note that the Movable concept does not require default construction. It is not difficult to conceive of algorithms that require both move semantics and default construction (i.e. any algorithm that declares uninitialized temporaries). We could extend the design by adding concepts for MoveSemiregular and MoveRegular that add default construction and equality comparison, respectively.

Despite the fact that we have added more concepts, we feel that this alternative is still well-within the ideals set forth in the introduction (§1). The Copyable and Movable concepts represent reasonably general ideas: types that can be copied or moved, respectively. We do not think that further fragmentation of these requirements would preserve the originally stated design ideals.

We recognize that these concepts could be further decomposed into smaller and smaller units. For example, we could easily see factoring out shared "object requirements" for the Copyable, Movable, and Function concepts. However, each subsequent refactoring yields concepts with less coherent meaning (if any). Any design resulting from this recursive fragmentation of requirements would fail to meet the design ideals stated in §1.3. We think that each step in breaking up larger semantic concepts into smaller language-specific ones should be taken with utmost care.

D.2 Cross-type Concepts

Cross-type concepts generalize a corresponding specific concept over different types. For example, the cross-type EqualityComparable concept describes the required syntax of comparing different types for equality and justifies the semantics of those operations by describing them in relation to the CommonType of its arguments.

There are some cases where the additional common type requirements are impractical. For example, the Boost Graph Library (Siek et al., 2001) defines a generic isomorphism algorithm on different graph types. Unfortunately, we could not encapsulate the algorithm as a Relation on different graph types because BGL graph types do not share a common type. Even though all graph data structures are obviously embedded in the same mathematical universe, there is no type to which both an adjacency list and adjacency matrix can be converted in order to

describe the semantics of relations on those different graphs. In these cases, the common type requirements are too restrictive; it even prevents us from using otherwise viable overloads.

An alternative approach to specifying cross-type operators and function objects is to relax the common type requirements by conditionally requiring them as part of the concept's semantics. Consider an alternative definition of EqualityComparable:

```
concept EqualityComparable<EqualityComparable T1, EqualityComparable T2> =
  requires (T1 a, T2 b) {
    bool { a == b };
    bool { b == a };

    axiom {
      if (Common<T1, T2> && EqualityComparable<CommonType<T1, T2>>) {
        a == b <=> C{a} == C{b};
        b == a <=> C{b} == C{a};
    }
  };
```

By moving the common type requirements into the axiom, we guarantee that the concept will be satisfied if there are available overloads for T1 and T2, regardless of the relationship between those types. We can only guarantee the meaning of the operation when T1 and T2 satisfy the additional requirements in the concept's axiom. Note that the semantics of Writable and MoveWritable are predicated in a similar manner.

Effectively, this definition makes the common type requirements "required to test" instead of "required to compile". That is, if the programmer wants to check the behavior of their operator against the common type, they will need to supply the necessary overloads.

The TotallyOrdered, Relation, and BinaryOperation could be modified in similar ways, although the impact on Relation and BinaryOperation may be significant §3.4. The current definition of Relation, for example, may require a programmer to write up to 12 overloads of operator() in extreme cases.

1. Two overloads for T1
2. Two overloads for T2
3. Two symmetric overloads for T1 and T2
4. Two overloads for CommonType<T1, T2>
5. Two symmetric overloads for T1 and CommonType<T1, T2>
6. Two symmetric overloads for T2 and CommonType<T1, T2>

The last 6 overloads would only be necessary if the CommonType of T1 and T2 was different than either T1 or T2, and the cross-type overloads (3, 5, and 6) would only need to be provided to avoid expensive conversions.

While this approach to defining cross-type concepts does reduce the burden of working with different types, it also means that programmers may instantiate algorithms over operations that have no sound mathematical meaning. This allows, for example, the use of == for employees and strings when the appropriate overloads are given. This is a trade-off, and one that we think fits within the C++ philosophy: don't make programmers pay for the features they don't use. Obviously, we would encourage programmers to write sound and verifiable code, but for the sake of compatibility, we can consider a design that relaxes the strictness of those requirements.

References

G. Dos Reis and B. Stroustrup. Specifying C++ concepts. In *Proc. 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 295–308. ACM Press, 2006.

G. Dos Reis, B. Stroustrup, and A. Merideth. Axioms: Semantics Aspects of C++ Concepts. Technical Report N2887=09-0077, ISO/IEC JTC1/SC22/WG21—The C++ Standards Committee, 2009.

D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: Linguistic support for generic programming in C++. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*, pages 291–310, Portland, Oregon, 2006.

C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12: 576–580,583, 1969.

J. Lakos. Normative Language to Describe Value Copy Semantics. Technical Report N2479=07-0349, ISO/IEC JTC1/SC22/WG21—The C++ Standards Committee, Dec. 2007.

Nokia Corporation. Qt, Dec. 2011. URL http://qt.nokia.com/products/.

J. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2001.

C++ Standards Committee. Working Draft, Standard for Programming Language C++. Technical Report N2914=09-0104, ISO/IEC JTC1/SC22/WG21—The C++ Standards Committee, June 2009.

C++ Standards Committee. Working Draft, Standard for Programming Language C++. Technical Report N3291=11-0061, ISO/IEC JTC1/SC22/WG21—The C++ Standards Committee, Apr. 2011.

A. Stepanov and P. McJones. *Elements of Programming*. Addison-Wesley, June 2009.

B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

B. Stroustrup. The c++0x 'remove concepts' decision, Aug. 2009.

A. Sutton. Origin c++ libraries, Dec. 2011. URL http://code.google.com/p/origin/.

A. Sutton and B. Stroustrup. Design of concept libraries for C++. In *Proc. SLE 2011*, July 2011.

D. van Heesch. Doxygen, Dec. 2011. URL http://www.stack.nl/~dimitri/doxygen/.

L. Voufo, M. Zalewski, and A. Lumsdaine. ConceptClang: An Implementation of C++ Concepts in Clang. In *Proc. 7th ACM SIGPLAN Workshop on Generic Programming*, pages 71–82. ACM Press, 2011.