The Standard Template Library

Alexander Stepanov

Silicon Graphics Inc.
2011 N. Shoreline Blvd.
Mt. View, CA 94043
stepanov@mti.sgi.com

Meng Lee

Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
lee@hpl.hp.com

October 31, 1995

1 Introduction

The Standard Template Library provides a set of well structured generic C++ components that work together in a seamless way. Special care has been taken to ensure that all the template algorithms work not only on the data structures in the library, but also on built-in C++ data structures. For example, all the algorithms work on regular pointers. The orthogonal design of the library allows programmers to use library data structures with their own algorithms, and to use library algorithms with their own data structures. The well specified semantic and complexity requirements guarantee that a user component will work with the library, and that it will work efficiently. This flexibility ensures the widespread utility of the library.

Another important consideration is efficiency. C++ is successful because it combines expressive power with efficiency. Much effort has been spent to verify that every template component in the library has a generic implementation that performs within a few percentage points of the efficiency of the corresponding hand coded routine.

The third consideration in the design has been to develop a library structure that, while being natural and easy to grasp, is based on a firm theoretical foundation.

2 Structure of the library

The library contains five main kinds of components:

- algorithm: defines a computational procedure.
- container: manages a set of memory locations.
- iterator: provides a means for an algorithm to traverse through a container.
- function object: encapsulates a function in an object for use by other components.
- adaptor: adapts a component to provide a different interface.

Such decomposition allows us to dramatically reduce the component space. For example, instead of providing a search member function for every kind of container we provide a single version that works with all of them as long as a basic set of requirements is satisfied.

The following description helps clarify the structure of the library. If software components are tabulated as a three-dimensional array, where one dimension represents different data types (e.g. int, double), the second dimension represents different containers (e.g. vector, linked-list, file), and the third dimension represents different algorithms on the containers (e.g. searching, sorting, rotation), if `i`, `j`, and `k` are the size of the dimensions, then `i*j*k` different versions of code have to be designed. By using template functions that are parameterized by a data type, we need only `j*k` versions. Further, by making our algorithms work on different containers, we need merely `j+k` versions. This significantly simplifies software design work and also makes it possible to use components in the library together with user defined components in a very flexible way. A user may easily define a specialized container class and use the library's sort function to sort it. A user may provide a different comparison function for the sort either as a regular pointer to a comparison function, or as a function object (an object with an `operator()` defined) that does the comparisons. If a user needs to iterate through a container in the reverse direction, the `reverse_iterator` adaptor allows that.

The library extends the basic C++ paradigms in a consistent way, so it is easy for a C/C++ programmer to start using the library. For example, the library contains a `merge` template function. When a user has two arrays `a` and `b` to be merged into `c` it can be done with:

```
int a[1000];
int b[2000];
int c[3000];
...
merge(a, a + 1000, b, b + 2000, c);
```

When a user wants to merge a vector and a list (both of which are template classes in the library) and put the result into a freshly allocated uninitialized storage it can be done with:

```
vector<Employee> a;
list<Employee> b;
...
Employee* c = allocate(a.size() + b.size(), (Employee*)0);
merge(a.begin(), a.end(), b.begin(), b.end(),
      raw_storage_iterator<Employee*, Employee>(c));
```

where `begin()` and `end()` are member functions of containers that return the right types of iterators or pointer-like objects that allow the merge to do the job and `raw_storage_iterator` is an adapter that allows algorithms to put results directly into uninitialized memory by calling the appropriate copy constructor.

In many cases it is useful to iterate through input/output streams in the same way as through regular data structures. For example, if we want to merge two data structures and then store them in a file, it would be nice to avoid creation of an auxiliary data structure for the result, instead storing the result directly into the corresponding file. The library provides both `istream_iterator` and `ostream_iterator` template classes to make many of the library algorithms work with I/O streams that represent homogenous aggregates of data. Here is a program that reads a file of integers from the standard input, removes all those that are divisible by its command argument, and writes the result to the standard output:

```
main(int argc, char** argv) {
      if (argc != 2) throw("usage: remove_if_divides integer\n");
      remove_copy_if(istream_iterator<int>(cin), istream_iterator<int>(),
            ostream_iterator<int>(cout, "\n"),
            not1(bind2nd(modulus<int>(), atoi(argv[1]))));
}
```

All the work is done by `remove_copy_if` which reads integers one by one until the input iterator becomes equal to the *end-of-stream* iterator that is constructed by the constructor with no arguments. (In general, all the algorithms work in a "from here to there" fashion taking two iterators that signify the beginning and the end of the input.) Then `remove_copy_if` writes the integers that pass the test onto the output stream through the output iterator that is bound to `cout`. As a predicate, `remove_copy_if` uses a function object constructed from a function object, `modulus<int>`, which takes `i` and `j` and returns `i%j`, as a binary predicate and makes it into a unary predicate by using `bind2nd` to bind the second argument to the command line argument, `atoi(argv[1])`. Then the negation of this unary predicate is obtained using function adaptor `not1`.

A somewhat more realistic example is a filter program that takes a file and randomly shuffles its lines.

```
main(int argc, char**) {
      if (argc != 1) throw("usage: shuffle\n");
      vector<string> v;
      copy(istream_iterator<string>(cin), istream_iterator<string>(),
            inserter(v, v.end()));
      random_shuffle(v.begin(), v.end());
      copy(v.begin(), v.end(), ostream_iterator<string>(cout));
}
```

In this example, `copy` moves lines from the standard input into a vector, but since the vector is not pre-allocated it uses an insert iterator to insert the lines one by one into the vector. (This technique allows all of the copying functions to work in the usual overwrite mode as well as in the insert mode.) Then `random_shuffle` shuffles the vector and another call to `copy` copies it onto the `cout` stream.

3 Requirements

To ensure that the different components in a library work together, they must satisfy some basic requirements. Requirements should be as general as possible, so instead of saying "class `X` has to define a member function `operator++()`," we say "for any object `x` of class `X`, `++x` is defined." (It is unspecified whether the operator is a member or a global function.) Requirements are stated in terms of well-defined expressions, which define valid terms of the types that satisfy the requirements. For every set of requirements there is a table that specifies an initial set of the valid expressions and their semantics. Any generic algorithm that uses the requirements has to be written in terms of the valid expressions for its formal type parameters.

If an operation is required to be linear time, it means no worse than linear time, and a constant time operation satisfies the requirement.

In some cases we present the semantic requirements using C++ code. Such code is intended as a specification of equivalence of a construct to another construct, not necessarily as the way the construct must be implemented (although in some cases the code given is unambiguously the optimum implementation).

4 Core components

This section contains some basic template functions and classes that are used throughout the rest of the library.

4.1 Operators

To avoid redundant definitions of `operator!=` out of `operator==` and `operators>`, `<=`, and `>=` out of `operator<` the library provides the following:

```
template <class T1, class T2>
inline bool operator!=(const T1& x, const T2& y) {
       return !(x == y);
}

template <class T1, class T2>
inline bool operator>(const T1& x, const T2& y) {
       return y < x;
}

template <class T1, class T2>
inline bool operator<=(const T1& x, const T2& y) {
       return !(y < x);
}

template <class T1, class T2>
inline bool operator>=(const T1& x, const T2& y) {
       return !(x < y);
}
```

4.2 Pair

The library includes templates for heterogeneous pairs of values.

```
template <class T1, class T2>
struct pair {
       T1 first;
       T2 second;
       pair() {}
       pair(const T1& x, const T2& y) : first(x), second(y) {}
};

template <class T1, class T2>
```

```
inline bool operator==(const pair<T1, T2>& x, const pair<T1, T2>& y) {
      return x.first == y.first && x.second == y.second;
}

template <class T1, class T2>
inline bool operator<(const pair<T1, T2>& x, const pair<T1, T2>& y) {
      return x.first < y.first || (!(y.first < x.first) && x.second < y.second);
}
```

The library provides a matching template function `make_pair` to simplify their construction. Instead of saying, for example,

```
return pair<int, double>(5, 3.1415926); // explicit types
```

one may say

```
return make_pair(5, 3.1415926); // types are deduced

template <class T1, class T2>
inline pair<T1, T2> make_pair(const T1& x, const T2& y) {
      return pair<T1, T2>(x, y);
```

5 Iterators

Iterators are a generalization of pointers that allow a programmer to work with different data structures (containers) in a uniform manner. To be able to construct template algorithms that work correctly and efficiently on different types of data structures, we need to formalize not just the interfaces but also the semantics and complexity assumptions of iterators. Iterators are objects that have `operator*` returning a value of some class or built-in type `T` called a *value type* of the iterator. For every iterator type `X` for which equality is defined, there is a corresponding signed integral type called the *distance type* of the iterator.

Since iterators are a generalization of pointers, their semantics is a generalization of the semantics of pointers in C++. This assures that every template function that takes iterators works with regular pointers. Depending on the operations defined on them, there are five categories of iterators: *input iterators, output iterators, forward iterators, bidirectional iterators* and *random access iterators*. Forward iterators satisfy all the requirements of the input and output iterators and can be used whenever either kind is specified. Bidirectional iterators satisfy all the requirements of the forward iterators and can be used whenever a forward iterator is specified. Random access iterators satisfy all the requirements of bidirectional iterators and can be used whenever a bidirectional iterator is specified. There is an additional attribute that forward, bidirectional and random access iterators might have, that is, they can be *mutable* or *constant* depending on whether the result of the `operator*` behaves as a reference or as a reference to a constant. Constant iterators do not satisfy the requirements for output iterators.

Table 1: Relations among iterator categories

Random access	$\longrightarrow$	Bidirectional	$\longrightarrow$	Forward	$\nearrow$	Input
					$\searrow$	Output

Just as a regular pointer to an array guarantees that there is a pointer value pointing past the last element of the array, so for any iterator type there is an iterator value that points past the last element of a corresponding container. These values are called *past-the-end* values. Values of the iterator for which the `operator*` is defined are called *dereferenceable*. The library never assumes that past-the-end values are dereferenceable. Iterators might also have *singular* values that are not associated with any container. For example, after the declaration of an uninitialized pointer `x` (as with `int* x;`), `x` should always be assumed to have a singular value of a pointer. Results of most expressions are undefined for singular values. The only exception is an assignment of a non-singular value to an iterator that holds a singular value. In this case the singular value is overwritten the same way as any other value. Dereferenceable and past-the-end values are always non-singular.

An iterator `j` is called *reachable* from an iterator `i` if and only if there is a finite sequence of applications of `operator++` to `i` that makes `i == j`. If `i` and `j` refer to the same container, then either `j` is reachable from `i`, or `i` is reachable from `j`, or both (`i == j`).

Most of the library's algorithmic templates that operate on data structures have interfaces that use ranges. A range is a pair of iterators that designate the beginning and end of the computation. A range `[i, i)` is an empty range; in general, a range `[i, j)` refers to the elements in the data structure starting with the one pointed to by `i` and up to but not including the one pointed to by `j`. Range `[i, j)` is valid if and only if `j` is reachable from `i`. The result of the application of the algorithms in the library to invalid ranges is undefined.

All the categories of iterators require only those functions that are realizable for a given category in constant time (amortized). Therefore, requirement tables for the iterators do not have a complexity column.

In the following sections, we assume: `a` and `b` are values of `X`, `n` is a value of the distance type `Distance`, `u`, `tmp`, and `m` are identifiers, `r` and `s` are lvalues of `X`, `t` is a value of value type `T`.

5.1 Input iterators

A class or a built-in type `X` satisfies the requirements of an input iterator for the value type `T` if the following expressions are valid:

Table 2: Input iterator requirements

expression	return type	operational semantics	assertion/note pre/post-condition
`X(a)`			`X(a)` is a copy of `a`. note: a destructor is assumed.
`X u(a);` `X u = a;`			post: `u` is a copy of `a`.
`u = a`	`X&`		post: `u` is a copy of `a`.
`a == b`	convertible to `bool`		if `a` is a copy of `b`, then `a == b` returns `true`. == is an equivalence relation over the domain of ==.
`a != b`	convertible to `bool`	`!(a == b)`	
`*a`	convertible to `T`		pre: `a` is dereferenceable. if `a` is a copy of `b`, then `*a` is equivalent to `*b`.
`++r`	`X&`		pre: `r` is dereferenceable. post: `r` is dereferenceable or `r` is past-the-end.
`(void)r++`	`void`	`(void)++r`	

Table 2: Input iterator requirements

expression	return type	operational semantics	assertion/note pre/post-condition
`*r++`	`T`	`{ X tmp = r; ++r; return tmp; }`	

NOTE: For input iterators, there are no requirements on the type or value of `r++` beyond the requirement that `*r++` works appropriately. In particular, `r == s` does not imply `++r == ++s`. (Equality does not guarantee the substitution property or referential transparency.) As for `++r`, there are no more requirements on the values of any copies of `r` except that they can be safely destroyed or assigned to. After executing `++r`, copies of (the previous) `r` are not required to be in the domain of `==`. Algorithms on input iterators should never attempt to pass through the same iterator twice. They should be *single pass* algorithms. *Value type `T` is not required to be an lvalue type*. These algorithms can be used with istreams as the source of the input data through the `istream_iterator` class.

5.2 Output iterators

A class or a built-in type `X` satisfies the requirements of an output iterator if the following expressions are valid:

Table 3: Output iterator requirements

expression	return type	operational semantics	assertion/note pre/post-condition
`X(a)`			`*a = t` is equivalent to `*X(a) = t`. note: a destructor is assumed.
`X u(a);` `X u = a;`			
`*a = t`	result is not used		
`++r`	`X&`		
`r++`	`X` or `X&`		

NOTE: The only valid use of an `operator*` is on the left side of the assignment statement. *Assignment through the same value of the iterator happens only once*. Algorithms on output iterators should never attempt to pass through the same iterator twice. They should be *single pass* algorithms. Equality and inequality are not necessarily defined. Algorithms that take output iterators can be used with ostreams as the destination for placing data through the `ostream_iterator` class as well as with insert iterators and insert pointers. In particular, the following two conditions should hold: first, any iterator value should be assigned through before it is incremented (this is, for an output iterator `i`, `i++; i++;` is not a valid code sequence); second, any value of an output iterator may have at most one active copy at any given time (for example, `i = j; *++i = a; *j = b;` is not a valid code sequence).

5.3 Forward iterators

A class or a built-in type `X` satisfies the requirements of a forward iterator if the following expressions are valid:

Table 4: Forward iterator requirements

expression	return type	operational semantics	assertion/note pre/post-condition
`X u;`			note: `u` might have a singular value. note: a destructor is assumed.
`X()`			note: `X()` might be singular.
`X(a)`			`a == X(a).`
`X u(a);` `X u = a;`		`X u; u = a;`	post: `u == a.`
`a == b`	convertible to `bool`		`==` is an equivalence relation.
`a != b`	convertible to `bool`	`!(a == b)`	
`r = a`	`X&`		post: `r == a.`
`*a`	convertible to `T`		pre: `a` is dereferenceable. `a == b` implies `*a == *b`. If `X` is mutable, `*a = t` is valid.
`++r`	`X&`		pre: `r` is dereferenceable. post: `r` is dereferenceable or `r` is past-the-end. `r == s` and `r` is dereferenceable implies `++r == ++s`. `&r == &++r.`
`r++`	`X`	`{ X tmp = r; ++r; return tmp; }`	

NOTE: The fact that `r == s` implies `++r == ++s` (which is not true for input and output iterators) and the removal on the restrictions on the number of the assignments through the iterator (which applies to output iterators) allows the use of multi-pass one-directional algorithms with forward iterators.

5.4 Bidirectional iterators

A class or a built-in type `X` satisfies the requirements of a bidirectional iterator if to the table that specifies forward iterators we add the following lines:

Table 5: Bidirectional iterator requirements (in addition to forward iterator)

expression	return type	operational semantics	assertion/note pre/post-condition

`--r`	`X&`		pre: there exists `s` such that `r == ++s.`
			post: `s` is dereferenceable.
			`--(++r) == r.`
			`--r == --s` implies `r == s.`
			`&r == &--r.`

Table 5: Bidirectional iterator requirements (in addition to forward iterator)

expression	return type	operational semantics	assertion/note pre/post-condition

`r--`	`X`	`{ X tmp = r;`
		`  --r;`
		`  return tmp; }`	

NOTE: Bidirectional iterators allow algorithms to move iterators backward as well as forward.

5.5 Random access iterators

A class or a built-in type `X` satisfies the requirements of a random access iterator if to the table that specifies bidirectional iterators we add the following lines:

Table 6: Random access iterator requirements (in addition to bidirectional iterator)

expression	return type	operational semantics	assertion/note pre/post-condition

`r += n`	`X&`	`{ Distance m = n;`
		`  if (m >= 0)`
		`    while (m--) ++r;`
		`  else`
		`    while (m++) --r;`
		`  return r; }`	

`a + n` `n + a`	`X`	`{ X tmp = a;` `  return tmp += n; }`	`a + n == n + a.`
`r -= n`	`X&`	`return r += -n;`	
`a - n`	`X`	`{ X tmp = a;` `  return tmp -= n; }`	
`b - a`	`Distance`		pre: there exists a value `n` of `Distance` such that `a + n = b`. `b == a + (b - a).`
`a[n]`	convertible to `T`	`*(a + n)`	
`a < b`	convertible to `bool`	`b - a > 0`	`<` is a total ordering relation
`a > b`	convertible to `bool`	`b < a`	`>` is a total ordering relation opposite to `<`.
`a >= b`	convertible to `bool`	`!(a < b)`	
`a <= b`	convertible to `bool`	`!(a > b)`	

5.6 Iterator tags

To implement algorithms only in terms of iterators, it is often necessary to infer both of the value type and the distance type from the iterator. To enable this task it is required that for an iterator `i` of any category other than output iterator, the expression `value_type(i)` returns `(T*)(0)` and the expression `distance_type(i)` returns `(Distance*)(0)`. For output iterators, these expressions are not required.

5.6.1 Examples of using iterator tags

For all the regular pointer types we can define `value_type` and `distance_type` with the help of:

```
template <class T>
inline T* value_type(const T*) { return (T*)(0); }

template <class T>
inline ptrdiff_t* distance_type(const T*) { return (ptrdiff_t*)(0); }
```

Then, if we want to implement a generic `reverse` function, we do the following:

```
template <class BidirectionalIterator>
inline void reverse(BidirectionalIterator first, BidirectionalIterator last) {
	__reverse(first, last, value_type(first), distance_type(first));
}
```

where `__reverse` is defined as:

```
template <class BidirectionalIterator, class T, class Distance>
void __reverse(BidirectionalIterator first, BidirectionalIterator last, T*,
               Distance*) {
	Distance n;
	distance(first, last, n); // see Iterator operations section
	--n;
	while (n > 0) {
		T tmp = *first;
		*first++ = *--last;
		*last = tmp;
		n -= 2;
	}
}
```

If there is an additional pointer type `__huge` such that the difference of two `__huge` pointers is of the type `long long`, we define:

```
template <class T>
inline T* value_type(const T __huge *) { return (T*)(0); }

template <class T>
inline long long* distance_type(const T __huge *) { return (long long*)(0); }
```

It is often desirable for a template function to find out what is the most specific category of its iterator argument, so that the function can select the most efficient algorithm at compile time. To facilitate this, the library introduces *category tag* classes which are used as compile time tags for algorithm selection. They are: `input_iterator_tag`, `output_iterator_tag`, `forward_iterator_tag`, `bidirectional_iterator_tag` and `random_access_iterator_tag`. Every iterator `i` must have an expression `iterator_category(i)` defined on it that returns the most specific category tag that describes its behavior. For example, we define that all the pointer types are in the random access iterator category by:

```
template <class T>
inline random_access_iterator_tag iterator_category(const T*) {
	return random_access_iterator_tag();
}
```

For a user-defined iterator `BinaryTreeIterator`, it can be included into the bidirectional iterator category by saying:

```
template <class T>
inline bidirectional_iterator_tag iterator_category(
```

```
                const BinaryTreeIterator<T>&) {
        return bidirectional_iterator_tag();
}
```

If a template function `evolve` is well defined for bidirectional iterators, but can be implemented more efficiently for random access iterators, then the implementation is like:

```
template <class BidirectionalIterator>
inline void evolve(BidirectionalIterator first, BidirectionalIterator last)
        evolve(first, last, iterator_category(first));
}

template <class BidirectionalIterator>
void evolve(BidirectionalIterator first, BidirectionalIterator last,
                bidirectional_iterator_tag) {
        // ... more generic, but less efficient algorithm
}

template <class RandomAccessIterator>
void evolve(RandomAccessIterator first, RandomAccessIterator last,
                random_access_iterator_tag) {
        // ... more efficient, but less generic algorithm
}
```

5.6.2 Library defined primitives

To simplify the task of defining the `iterator_category`, `value_type` and `distance_type` for user definable iterators, the library provides the following predefined classes and functions:

```
// iterator tags

struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag {};
struct bidirectional_iterator_tag {};
struct random_access_iterator_tag {};

// iterator bases

template <class T, class Distance = ptrdiff_t> struct input_iterator {};
struct output_iterator {};
// output_iterator is not a template because output iterators
// do not have either value type or distance type defined.
template <class T, class Distance = ptrdiff_t> struct forward_iterator {};
template <class T, class Distance = ptrdiff_t> struct bidirectional_iterator {};
template <class T, class Distance = ptrdiff_t> struct random_access_iterator {};

// iterator_category

template <class T, class Distance>
inline input_iterator_tag
iterator_category(const input_iterator<T, Distance>&) {
        return input_iterator_tag();
}
inline output_iterator_tag iterator_category(const output_iterator&) {
        return output_iterator_tag();
}
template <class T, class Distance>
inline forward_iterator_tag
```

```
iterator_category(const forward_iterator<T, Distance>&) {
       return forward_iterator_tag();
}
template <class T, class Distance>
inline bidirectional_iterator_tag
iterator_category(const bidirectional_iterator<T, Distance>&) {
       return bidirectional_iterator_tag();
}
template <class T, class Distance>
inline random_access_iterator_tag
iterator_category(const random_access_iterator<T, Distance>&) {
       return random_access_iterator_tag();
}
template <class T>
inline random_access_iterator_tag iterator_category(const T*) {
       return random_access_iterator_tag();
}

// value_type of iterator

template <class T, class Distance>
inline T* value_type(const input_iterator<T, Distance>&) {
       return (T*)(0);
}
template <class T, class Distance>
inline T* value_type(const forward_iterator<T, Distance>&) {
       return (T*)(0);
}
template <class T, class Distance>
inline T* value_type(const bidirectional_iterator<T, Distance>&) {
       return (T*)(0);
}
template <class T, class Distance>
inline T* value_type(const random_access_iterator<T, Distance>&) {
       return (T*)(0);
}
template <class T>
inline T* value_type(const T*) { return (T*)(0); }

// distance_type of iterator

template <class T, class Distance>
inline Distance* distance_type(const input_iterator<T, Distance>&) {
       return (Distance*)(0);
}
template <class T, class Distance>
inline Distance* distance_type(const forward_iterator<T, Distance>&) {
       return (Distance*)(0);
}
template <class T, class Distance>
inline Distance* distance_type(const bidirectional_iterator<T, Distance>&) {
       return (Distance*)(0);
}
template <class T, class Distance>
inline Distance* distance_type(const random_access_iterator<T, Distance>&) {
       return (Distance*)(0);
}
template <class T>
inline ptrdiff_t* distance_type(const T*) { return (ptrdiff_t*)(0); }
```

If a user wants to define a bidirectional iterator for some data structure containing `double` and such that it works on a large memory model of a computer, it can be done by defining:

```
class MyIterator : public bidirectional_iterator<double, long> {
	// code implementing ++, etc.
};
```

Then there is no need to define `iterator_category`, `value_type`, and `distance_type` on `MyIterator`.

5.7 Iterator operations

Since only random access iterators provide + and - operators, the library provides two template functions `advance` and `distance`. These functions use + and - for random access iterators (and are, therefore, constant time for them); for input, forward and bidirectional iterators they use ++ to provide linear time implementations. `advance` takes a negative argument `n` for random access and bidirectional iterators only. `advance` increments (or decrements for negative `n`) iterator reference `i` by `n`. `distance` increments `n` by the number of times it takes to get from `first` to `last`.

```
template <class InputIterator, class Distance>
inline void advance(InputIterator& i, Distance n);

template <class InputIterator, class Distance>
inline void distance(InputIterator first, InputIterator last, Distance& n);
```

`distance` must be a three argument function storing the result into a reference instead of returning the result because the distance type cannot be deduced from built-in iterator types such as `int*`.

6 Function objects

Function objects are objects with an `operator()` defined. They are important for the effective use of the library. In the places where one would expect to pass a pointer to a function to an algorithmic template, the interface is specified to accept an object with an `operator()` defined. This not only makes algorithmic templates work with pointers to functions, but also enables them to work with arbitrary function objects. Using function objects together with function templates increases the expressive power of the library as well as making the resulting code much more efficient. For example, if we want to have a by-element addition of two vectors `a` and `b` containing `double` and put the result into `a` we can do:

```
transform(a.begin(), a.end(), b.begin(), a.begin(), plus<double>());
```

If we want to negate every element of `a` we can do:

```
transform(a.begin(), a.end(), a.begin(), negate<double>());
```

The corresponding functions will inline the addition and the negation.

To enable adaptors and other components to manipulate function objects that take one or two arguments it is required that they correspondingly provide typedefs `argument_type` and `result_type` for function objects that take one argument and `first_argument_type`, `second_argument_type`, and `result_type` for function objects that take two arguments.

6.1 Base

The following classes are provided to simplify the typedefs of the argument and result types:

```
template <class Arg, class Result>
struct unary_function {
	typedef Arg argument_type;
	typedef Result result_type;
};
```

```
template <class Arg1, class Arg2, class Result>
struct binary_function {
      typedef Arg1 first_argument_type;
      typedef Arg2 second_argument_type;
      typedef Result result_type;
};
```

6.2 Arithmetic operations

The library provides basic function object classes for all of the arithmetic operators in the language.

```
template <class T>
struct plus : binary_function<T, T, T> {
      T operator()(const T& x, const T& y) const { return x + y; }
};

template <class T>
struct minus : binary_function<T, T, T> {
      T operator()(const T& x, const T& y) const { return x - y; }
};

template <class T>
struct times : binary_function<T, T, T> {
      T operator()(const T& x, const T& y) const { return x * y; }
};

template <class T>
struct divides : binary_function<T, T, T> {
      T operator()(const T& x, const T& y) const { return x / y; }
};

template <class T>
struct modulus : binary_function<T, T, T> {
      T operator()(const T& x, const T& y) const { return x % y; }
};

template <class T>
struct negate : unary_function<T, T> {
      T operator()(const T& x) const { return -x; }
};
```

6.3 Comparisons

The library provides basic function object classes for all of the comparison operators in the language.

```
template <class T>
struct equal_to : binary_function<T, T, bool> {
      bool operator()(const T& x, const T& y) const { return x == y; }
};

template <class T>
struct not_equal_to : binary_function<T, T, bool> {
      bool operator()(const T& x, const T& y) const { return x != y; }
};

template <class T>
struct greater : binary_function<T, T, bool> {
      bool operator()(const T& x, const T& y) const { return x > y; }
```

```
};

template <class T>
struct less : binary_function<T, T, bool> {
      bool operator()(const T& x, const T& y) const { return x < y; }
};

template <class T>
struct greater_equal : binary_function<T, T, bool> {
      bool operator()(const T& x, const T& y) const { return x >= y; }
};

template <class T>
struct less_equal : binary_function<T, T, bool> {
      bool operator()(const T& x, const T& y) const { return x <= y; }
};
```

6.4 Logical operations

```
template <class T>
struct logical_and : binary_function<T, T, bool> {
      bool operator()(const T& x, const T& y) const { return x && y; }
};

template <class T>
struct logical_or : binary_function<T, T, bool> {
      bool operator()(const T& x, const T& y) const { return x || y; }
};

template <class T>
struct logical_not : unary_function<T, bool> {
      bool operator()(const T& x) const { return !x; }
};
```

7 Allocators

One of the common problems in portability is to be able to encapsulate the information about the memory model. This information includes the knowledge of pointer types, the type of their difference, the type of the size of objects in this memory model, as well as the memory allocation and deallocation primitives for it.

STL addresses this problem by providing a standard set of requirements for *allocators*, which are objects that encapsulate this information. All of the containers in STL are parameterized in terms of allocators. That dramatically simplifies the task of dealing with multiple memory models.

7.1 Allocator requirements

In the following table, we assume `X` is an allocator class for objects of type `T`, `a` is a value of `X`, `n` is of type `X::size_type`, `p` is of type `X::pointer`, `r` is of type `X::reference` and `s` is of type `X::const_reference`.

All the operations on the allocators are expected to be amortized constant time.

Table 7: Allocator requirements

expression	return type	assertion/note pre/post-condition
`X::value_type`	`T`	

Table 7: Allocator requirements

expression	return type	assertion/note pre/post-condition
X::reference	lvalue of T	
X::const_reference	const lvalue of T	
X::pointer	pointer to T type	the result of operator* of values of X::pointer is of reference.
X::const_pointer	pointer to const T type	the result of operator* of values of X::const_pointer is of const_reference; it is the same type of pointer as X::pointer, in particular, sizeof(X::const_pointer) == sizeof(X::pointer).
X::size_type	unsigned integral type	the type that can represent the size of the largest object in the memory model.
X::difference_type	signed integral type	the type that can represent the difference between any two pointers in the memory model.
X a;		note: a destructor is assumed.
a.address(r)	pointer	*(a.address(r)) == r.
a.const_address(s)	const_pointer	*(a.address(s)) == s.
a.allocate(n)	X::pointer	memory is allocated for n objects of type T but objects are not constructed. allocate may raise an appropriate exception.
a.deallocate(p)	result is not used	all the objects in the area pointed by p should be destroyed prior to the call of the deallocate.
construct(p, a)	void	post: *p == a.
destroy(p)	void	the value pointed by p is destroyed.
a.init_page_size()	X::size_type	the returned value is the optimal value for an initial buffer size of the given type. It is assumed that if k is returned by init_page_size, t is the construction time for T, and u is the time that it takes to do allocate(k), then k * t is much greater than u.
a.max_size()	X::size_type	the largest positive value of X::difference_type

pointer belongs to the category of mutable random access iterators referring to T. const_pointer belongs to the category of constant random access iterators referring to T. There is a conversion defined from pointer to const_pointer.

For any allocator template Alloc there is a specialization for type void. Alloc<void> has only constructor, destructor, and Alloc<void>::pointer defined. Conversions are defined from any instance of Alloc<T>::pointer into Alloc<void>::pointer and back so that for any p, p == Alloc<T>::pointer(Alloc<void>::pointer(p)).

7.2 The default allocator

```
template <class T>
class allocator {
public:
        typedef T* pointer;
        typedef const T* const_pointer;
        typedef T& reference;
        typedef const T& const_reference;
        typedef T value_type;
        typedef size_t size_type;
        typedef ptrdiff_t difference_type;
        allocator();
        ~allocator();
        pointer address(reference x);
        const_pointer const_address(const_reference x);
        pointer allocate(size_type n);
        void deallocate(pointer p);
        size_type init_page_size();
        size_type max_size();
};

class allocator<void> {
public:
        typedef void* pointer;
        allocator();
        ~allocator();
};
```

In addition to `allocator` the library vendors are expected to provide allocators for all supported memory models.

8 Containers

Containers are objects that store other objects. They control allocation and deallocation of these objects through constructors, destructors, insert and erase operations.

In the following table, we assume `X` is a container class containing objects of type `T`, `a` and `b` are values of `X`, `u` is an identifier and `r` is a value of `X&`.

Table 8: Container requirements

expression	return type	operational semantics	assertion/note pre/post-condition	complexity
`X::value_type`	`T`			compile time
`X::reference`				compile time
`X::const_reference`				compile time
`X::pointer`	a pointer type pointing to `X::reference`		pointer to `T` in the memory model used by the container	compile time

Table 8: Container requirements

expression	return type	operational semantics	assertion/note pre/post-condition	complexity
`X::iterator`	iterator type pointing to `X::reference`		an iterator of any iterator category except output iterator.	compile time
`X::const_iterator`	iterator type pointing to `X::const_reference`		a constant iterator of any iterator category except output iterator.	compile time
`X::difference_type`	signed integral type		is identical to the distance type of `X::iterator` and `X::const_iterator`	compile time
`X::size_type`	unsigned integral type		`size_type` can represent any non-negative value of `difference_type`	compile time
`X u;`			post: `u.size() == 0`.	constant
`X()`			`X().size() == 0`.	constant
`X(a)`			`a == X(a)`.	linear
`X u(a);` `X u = a;`		`X u; u = a;`	post: `u == a`.	linear
`(&a)->~X()`	result is not used		post: `a.size() == 0`. note: the destructor is applied to every element of `a` and all the memory is returned.	linear
`a.begin()`	`iterator`; `const_iterator` for constant `a`			constant
`a.end()`	`iterator`; `const_iterator` for constant `a`			constant
`a == b`	convertible to `bool`	`a.size() == b.size() && equal(a.begin(), a.end(), b.begin())`	`==` is an equivalence relation. note: `equal` is defined in the algorithms section.	linear
`a != b`	convertible to `bool`	`!(a == b)`		linear
`r = a`	`X&`	`if (&r != &a) { (&r)->X::~X(); new (&r) X(a); return r; }`	post: `r == a`.	linear

Table 8: Container requirements

expression	return type	operational semantics	assertion/note pre/post-condition	complexity
`a.size()`	`size_type`	`size_type n = 0; distance (a.begin(), a.end(), n); return n;`		constant
`a.max_size()`	`size_type`		`size()` of the largest possible container.	constant
`a.empty()`	convertible to `bool`	`a.size() == 0`		constant
`a < b`	convertible to `bool`	`lexicographical_compare(a.begin(), a.end(), b.begin(), b.end())`	pre: < is defined for values of `T`. < is a total ordering relation. `lexicographical_compare` is defined in the algorithms section.	linear
`a > b`	convertible to `bool`	`b < a`		linear
`a <= b`	convertible to `bool`	`!(a > b)`		linear
`a >= b`	convertible to `bool`	`!(a < b)`		linear
`a.swap(b)`	`void`	`swap(a,b)`		constant

The member function `size()` returns the number of elements in the container. Its semantics is defined by the rules of constructors, inserts, and erases.

`begin()` returns an iterator referring to the first element in the container. `end()` returns an iterator which is the past-the-end value.

If the iterator type of a container belongs to the bidirectional or random access iterator categories, the container is called `reversible` and satisfies the following additional requirements:

Table 9: Reversible container requirements (in addition to container)

expression	return type	operational semantics	complexity
`X::reverse_iterator`		`reverse_iterator<iterator, value_type, reference, difference_type>` for random access iterator `reverse_bidirectional_iterator<iterator, value_type, reference, difference_type>` for bidirectional iterator	compile time

Table 9: Reversible container requirements (in addition to container)

expression	return type	operational semantics	complexity
X::const_reverse_iterator		reverse_iterator<const_iterator, value_type, const_reference, difference_type> for random access iterator reverse_bidirectional_iterator<const_iterator, value_type, const_reference, difference_type> for bidirectional iterator	compile time
a.rbegin()	reverse_iterator; const_reverse_iterator for constant a	reverse_iterator(end())	constant
a.rend()	reverse_iterator; const_reverse_iterator for constant a	reverse_iterator(begin())	constant

8.1 Sequences

A sequence is a kind of container that organizes a finite set of objects, all of the same type, into a strictly linear arrangement. The library provides three basic kinds of sequence containers: vector, list, and deque. It also provides container adaptors that make it easy to construct abstract data types, such as stacks or queues, out of the basic sequence kinds (or out of other kinds of sequences that the user might define).

In the following two tables, X is a sequence class, a is value of X, i and j satisfy input iterator requirements, [i, j) is a valid range, n is a value of X::size_type, p is a valid iterator to a, q is a dereferenceable iterator to a, [q1, q2) is a valid range in a, t is a value of X::value_type.

The complexities of the expressions are sequence dependent.

Table 10: Sequence requirements (in addition to container)

expression	return type	assertion/note pre/post-condition
X(n, t) X a(n, t);		post: size() == n. constructs a sequence with n copies of t.
X(i, j) X a(i, j);		post: size() == distance between i and j. constructs a sequence equal to the range [i, j).
a.insert(p, t)	iterator	inserts a copy of t before p. the return value points to the inserted copy.
a.insert(p, n, t)	result is not used	inserts n copies of t before p.
a.insert(p, i, j)	result is not used	inserts copies of elements in [i, j) before p.
a.erase(q)	result is not used	erases the element pointed to by q.
a.erase(q1, q2)	result is not used	erases the elements in the range [q1, q2).

vector, list, and deque offer the programmer different complexity trade-offs and should be used accordingly. vector is the type of sequence that should be used by default. list should be used when there are frequent insertions and deletions from the middle of the sequence. deque is the data structure of choice when most insertions and deletions take place at the beginning or at the end of the sequence.

iterator and const_iterator types for sequences have to be at least of the forward iterator category.

Table 11: Optional sequence operations

expression	return type	operational semantics	container
a.front()	reference; const_reference for constant a	*a.begin()	vector, list, deque
a.back()	reference; const_reference for constant a	*a.(--end())	vector, list, deque
a.push_front(t)	void	a.insert(a.begin(), t)	list, deque
a.push_back(t)	void	a.insert(a.end(), t)	vector, list, deque
a.pop_front()	void	a.erase(a.begin())	list, deque
a.pop_back()	void	a.erase(--a.end())	vector, list, deque
a[n]	reference; const_reference for constant a	*(a.begin() + n)	vector, deque

All the operations in the above table are provided only for the containers for which they take constant time.

8.1.1 Vector

vector is a kind of sequence that supports random access iterators. In addition, it supports (amortized) constant time insert and erase operations at the end; insert and erase in the middle take linear time. Storage management is handled automatically, though hints can be given to improve efficiency.

```
template <class T, template <class U> class Allocator = allocator>
class vector {
public:

// typedefs:

      typedef iterator;
      typedef const_iterator;
      typedef Allocator<T>::pointer pointer;
      typedef Allocator<T>::reference reference;
      typedef Allocator<T>::const_reference const_reference;
      typedef size_type;
      typedef difference_type;
      typedef T value_type;
      typedef reverse_iterator;
      typedef const_reverse_iterator;

// allocation/deallocation:
```

```
        vector();
        vector(size_type n, const T& value = T());
        vector(const vector<T, Allocator>& x);
        template <class InputIterator>
        vector(InputIterator first, InputIterator last);
        ~vector();
        vector<T, Allocator>& operator=(const vector<T, Allocator>& x);
        void reserve(size_type n);
        void swap(vector<T, Allocator>& x);

// accessors:

        iterator begin();
        const_iterator begin() const;
        iterator end();
        const_iterator end() const;
        reverse_iterator rbegin();
        const_reverse_iterator rbegin();
        reverse_iterator rend();
        const_reverse_iterator rend();
        size_type size() const;
        size_type max_size() const;
        size_type capacity() const;
        bool empty() const;
        reference operator[](size_type n);
        const_reference operator[](size_type n) const;
        reference front();
        const_reference front() const;
        reference back();
        const_reference back() const;

// insert/erase:

        void push_back(const T& x);
        iterator insert(iterator position, const T& x = T());
        void insert(iterator position, size_type n, const T& x);
        template <class InputIterator>
        void insert(iterator position, InputIterator first, InputIterator last);
        void pop_back();
        void erase(iterator position);
        void erase(iterator first, iterator last);
};

template <class T, class Allocator>
bool operator==(const vector<T, Allocator>& x, const vector<T, Allocator>& y);

template <class T, class Allocator>
bool operator<(const vector<T, Allocator>& x, const vector<T, Allocator>& y);
```

`iterator` is a random access iterator referring to `T`. The exact type is implementation dependent and determined by `Allocator`.

`const_iterator` is a constant random access iterator referring to `const T`. The exact type is implementation dependent and determined by `Allocator`. It is guaranteed that there is a constructor for `const_iterator` out of `iterator`.

`size_type` is an unsigned integral type. The exact type is implementation dependent and determined by `Allocator`.

`difference_type` is a signed integral type. The exact type is implementation dependent and determined by `Allocator`.

The constructor `template <class InputIterator> vector(InputIterator first, InputIterator last)` makes only `N` calls to the copy constructor of `T` (where `N` is the distance between `first` and `last`) and no reallocations if iterators `first` and `last` are of forward, bidirectional, or random access categories. It does at most `2N` calls to the copy constructor of `T` and `logN` reallocations if they are just input iterators, since it is impossible to determine the distance between `first` and `last` and then do copying.

The member function `capacity` returns the size of the allocated storage in the vector. The member function `reserve` is a directive that informs `vector` of a planned change in size, so that it can manage the storage allocation accordingly. It does not change the size of the sequence and takes at most linear time in the size of the sequence. Reallocation happens at this point if and only if the current capacity is less than the argument of `reserve`. After `reserve`, `capacity` is greater or equal to the argument of `reserve` if reallocation happens; and equal to the previous value of `capacity` otherwise. Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence. It is guaranteed that no reallocation takes place during the insertions that happen after `reserve` takes place till the time when the size of the vector reaches the size specified by `reserve`.

`insert` causes reallocation if the new size is greater than the old capacity. If no reallocation happens, all the iterators and references before the insertion point remain valid. Inserting a single element into a vector is linear in the distance from the insertion point to the end of the vector. The amortized complexity over the lifetime of a vector of inserting a single element at its end is constant. Insertion of multiple elements into a vector with a single call of the insert member function is linear in the sum of the number of elements plus the distance to the end of the vector. In other words, it is much faster to insert many elements into the middle of a vector at once than to do the insertion one at a time. The insert template member function preallocates enough storage for the insertion if the iterators `first` and `last` are of forward, bidirectional or random access category. Otherwise, it does insert elements one by one and should not be used for inserting into the middle of vectors.

`erase` invalidates all the iterators and references after the point of the erase. The destructor of `T` is called the number of times equal to the number of the elements erased, but the assignment operator of `T` is called the number of times equal to the number of elements in the vector after the erased elements.

To optimize space allocation, a specialization for `bool` is provided:

```
class vector<bool, allocator> {
public:

// bit reference:

        class reference {
        public:
                ~reference();
                operator bool() const;
                reference& operator=(const bool x);
                void flip();        // flips the bit
        };

// typedefs:

        typedef bool const_reference;
        typedef iterator;
        typedef const_iterator;
        typedef size_t size_type;
```

```
        typedef ptrdiff_t difference_type;
        typedef bool value_type;
        typedef reverse_iterator;
        typedef const_reverse_iterator;

// allocation/deallocation:

        vector();
        vector(size_type n, const bool& value = bool());
        vector(const vector<bool, allocator>& x);
        template <class InputIterator>
        vector(InputIterator first, InputIterator last);
        ~vector();
        vector<bool, allocator>& operator=(const vector<bool, allocator>& x);
        void reserve(size_type n);
        void swap(vector<bool, allocator>& x);

// accessors:

        iterator begin();
        const_iterator begin() const;
        iterator end();
        const_iterator end() const;
        reverse_iterator rbegin();
        const_reverse_iterator rbegin();
        reverse_iterator rend();
        const_reverse_iterator rend();
        size_type size() const;
        size_type max_size() const;
        size_type capacity() const;
        bool empty() const;
        reference operator[](size_type n);
        const_reference operator[](size_type n) const;
        reference front();
        const_reference front() const;
        reference back();
        const_reference back() const;

// insert/erase:

        void push_back(const bool& x);
        iterator insert(iterator position, const bool& x = bool());
        void insert (iterator position, size_type n, const bool& x);
        template <class InputIterator>
        void insert (iterator position, InputIterator first, InputIterator last);
        void pop_back();
        void erase(iterator position);
        void erase(iterator first, iterator last);
};

void swap(vector<bool, allocator>::reference x,
         vector<bool, allocator>::reference y);

bool operator==(const vector<bool, allocator>& x,
               const vector<bool, allocator>& y);

bool operator<(const vector<bool, allocator>& x,
               const vector<bool, allocator>& y);
```

`reference` is a class that simulates the behavior of references of a single bit in `vector<bool>`.

Every implementation is expected to provide specializations of `vector<bool>` for all supported memory models.

At present, it is not possible to templatize a specialization. That is, we cannot write:

```
template <template <class U> class Allocator = allocator>
class vector<bool, Allocator> { /* ... */ };
```

Therefore, only `vector<bool, allocator>` is provided.

8.1.2 List

`list` is a kind of sequence that supports bidirectional iterators and allows constant time insert and erase operations anywhere within the sequence, with storage management handled automatically. Unlike vectors and deques, fast random access to list elements is not supported, but many algorithms only need sequential access anyway.

```
template <class T, template <class U> class Allocator = allocator>
class list {
public:

// typedefs:

    typedef iterator;
    typedef const_iterator;
    typedef Allocator<T>::pointer pointer;
    typedef Allocator<T>::reference reference;
    typedef Allocator<T>::const_reference const_reference;
    typedef size_type;
    typedef difference_type;
    typedef T value_type;
    typedef reverse_iterator;
    typedef const_reverse_iterator;

// allocation/deallocation:

    list();
    list(size_type n, const T& value = T());
    template <class InputIterator>
    list(InputIterator first, InputIterator last);
    list(const list<T, Allocator>& x);
    ~list();
    list<T, Allocator>& operator=(const list<T, Allocator>& x);
    void swap(list<T, Allocator>& x);

// accessors:

    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin();
```

```
        reverse_iterator rend();
        const_reverse_iterator rend();
        bool empty() const;
        size_type size() const;
        size_type max_size() const;
        reference front();
        const_reference front() const;
        reference back();
        const_reference back() const;

// insert/erase:

        void push_front(const T& x);
        void push_back(const T& x);
        iterator insert(iterator position, const T& x = T());
        void insert(iterator position, size_type n, const T& x);
        template <class InputIterator>
        void insert(iterator position, InputIterator first, InputIterator last);
        void pop_front();
        void pop_back();
        void erase(iterator position);
        void erase(iterator first, iterator last);

// special mutative operations on list:

        void splice(iterator position, list<T, Allocator>& x);
        void splice(iterator position, list<T, Allocator>& x, iterator i);
        void splice(iterator position, list<T, Allocator>& x, iterator first,
                    iterator last);
        void remove(const T& value);
        template <class Predicate> void remove_if(Predicate pred);
        void unique();
        template <class BinaryPredicate> void unique(BinaryPredicate binary_pred);
        void merge(list<T, Allocator>& x);
        template <class Compare> void merge(list<T, Allocator>& x, Compare comp);
        void reverse();
        void sort();
        template <class Compare> void sort(Compare comp);
};

template <class T, class Allocator>
bool operator==(const list<T, Allocator>& x, const list<T, Allocator>& y);

template <class T, class Allocator>
bool operator<(const list<T, Allocator>& x, const list<T, Allocator>& y);
```

`iterator` is a bidirectional iterator referring to `T`. The exact type is implementation dependent and determined by `Allocator`.

`const_iterator` is a constant bidirectional iterator referring to `const T`. The exact type is implementation dependent and determined by `Allocator`. It is guaranteed that there is a constructor for `const_iterator` out of `iterator`.

`size_type` is an unsigned integral type. The exact type is implementation dependent and determined by `Allocator`.

`difference_type` is a signed integral type. The exact type is implementation dependent and determined by `Allocator`.

`insert` does not affect the validity of iterators and references. Insertion of a single element into a list takes constant time and exactly one call to the copy constructor of `T`. Insertion of multiple elements into a list is linear in the number of elements inserted, and the number of calls to the copy constructor of `T` is exactly equal to the number of elements inserted.

`erase` invalidates only the iterators and references to the erased elements. Erasing a single element is a constant time operation with a single call to the destructor of `T`. Erasing a range in a list is linear time in the size of the range and the number of calls to the destructor of type `T` is exactly equal to the size of the range.

Since lists allow fast insertion and erasing from the middle of a list, certain operations are provided specifically for them:

`list` provides three splice operations that destructively move elements from one list to another:

`void splice(iterator position, list<T, Allocator>& x)` inserts the contents of `x` before `position` and `x` becomes empty. It takes constant time. The result is undefined if `&x == this`.

`void splice(iterator position, list<T, Allocator>& x, iterator i)` inserts an element pointed to by `i` from list `x` before `position` and removes the element from `x`. It takes constant time. `i` is a valid dereferenceable iterator of `x`. The result is unchanged if `position == i` or `position == ++i`.

`void splice(iterator position, list<T, Allocator>& x, iterator first, iterator last)` inserts elements in the range `[first, last)` before `position` and removes the elements from `x`. It takes constant time if `&x == this`; otherwise, it takes linear time. `[first, last)` is a valid range in `x`. The result is undefined if `position` is an iterator in the range `[first, last)`.

`remove` erases all the elements in the list referred by the list iterator `i` for which the following conditions hold: `*i == value`, `pred(*i) == true`. `remove` is stable, that is, the relative order of the elements that are not removed is the same as their relative order in the original list. Exactly `size()` applications of the corresponding predicate are done.

`unique` erases all but the first element from every consecutive group of equal elements in the list. Exactly `size() - 1` applications of the corresponding binary predicate are done.

`merge` merges the argument list into the list (both are assumed to be sorted). The merge is stable, that is, for equal elements in the two lists, the elements from the list always precede the elements from the argument list. `x` is empty after the merge. At most `size() + x.size() - 1` comparisons are done.

`reverse` reverses the order of the elements in the list. It is linear time.

`sort` sorts the list according to the `operator<` or a compare function object. It is stable, that is, the relative order of the equal elements is preserved. Approximately `NlogN` comparisons are done where `N` is equal to `size()`.

8.1.3 Deque

`deque` is a kind of sequence that, like a vector, supports random access iterators. In addition, it supports constant time insert and erase operations at the beginning or the end; insert and erase in the middle take linear time. As with vectors, storage management is handled automatically.

```
template <class T, template <class U> class Allocator = allocator>
class deque {
public:

// typedefs:

    typedef iterator;
    typedef const_iterator;
    typedef Allocator<T>::pointer pointer;
    typedef Allocator<T>::reference reference;
```

```
        typedef Allocator<T>::const_reference const_reference;
        typedef size_type;
        typedef difference_type;
        typedef T value_type;
        typedef reverse_iterator;
        typedef const_reverse_iterator;

// allocation/deallocation:

        deque();
        deque(size_type n, const T& value = T());
        deque(const deque<T, Allocator>& x);
        template <class InputIterator>
        deque(InputIterator first, InputIterator last);
        ~deque();
        deque<T, Allocator>& operator=(const deque<T, Allocator>& x);
        void swap(deque<T, Allocator>& x);

// accessors:

        iterator begin();
        const_iterator begin() const;
        iterator end();
        const_iterator end() const;
        reverse_iterator rbegin();
        const_reverse_iterator rbegin();
        reverse_iterator rend();
        const_reverse_iterator rend();
        size_type size() const;
        size_type max_size() const;
        bool empty() const;
        reference operator[](size_type n);
        const_reference operator[](size_type n) const;
        reference front();
        const_reference front() const;
        reference back();
        const_reference back() const;

// insert/erase:

        void push_front(const T& x);
        void push_back(const T& x);
        iterator insert(iterator position, const T& x = T());
        void insert (iterator position, size_type n, const T& x);
        template <class InputIterator>
        void insert (iterator position, InputIterator first, InputIterator last);
        void pop_front();
        void pop_back();
        void erase(iterator position);
        void erase(iterator first, iterator last);
};

template <class T, class Allocator>
bool operator==(const deque<T, Allocator>& x, const deque<T, Allocator>& y);

template <class T, class Allocator>
bool operator<(const deque<T, Allocator>& x, const deque<T, Allocator>& y);
```

iterator is a random access iterator referring to T. The exact type is implementation dependent and determined by Allocator.

const_iterator is a constant random access iterator referring to const T. The exact type is implementation dependent and determined by Allocator. It is guaranteed that there is a constructor for const_iterator out of iterator.

size_type is an unsigned integral type. The exact type is implementation dependent and determined by Allocator.

difference_type is a signed integral type. The exact type is implementation dependent and determined by Allocator.

insert in the middle of a deque invalidates all the iterators and references to the deque. insert and push at either end of a deque invalidate all the iterators to the deque, but have no effect on the validity of all the references to the deque. In the worst case, inserting a single element into a deque takes time linear in the minimum of the distance from the insertion point to the beginning of the deque and the distance from the insertion point to the end of the deque. Inserting a single element either at the beginning or end of a deque always takes constant time and causes a single call to the copy constructor of T. That is, a deque is especially optimized for pushing and popping elements at the beginning and end.

erase in the middle of a deque invalidates all the iterators and references to the deque. erase and pop at either end of a deque invalidate only the iterators and the references to the erased element. The number of calls to the destructor is the same as the number of elements erased, but the number of the calls to the assignment operator is equal to the minimum of the number of elements before the erased elements and the number of element after the erased elements.

8.2 Associative containers

Associative containers provide an ability for fast retrieval of data based on keys. The library provides four basic kinds of associative containers: set, multiset, map and multimap.

All of them are parameterized on Key and an ordering relation Compare that induces a total ordering on elements of Key. In addition, map and multimap associate an arbitrary type T with the Key. The object of type Compare is called the *comparison object* of a container.

In this section when we talk about equality of keys we mean the equivalence relation imposed by the comparison and *not* the operator== on keys. That is, two keys k1 and k2 are considered to be equal if for the comparison object comp, comp(k1, k2) == false && comp(k2, k1) == false.

An associative container supports *unique keys* if it may contain at most one element for each key. Otherwise, it supports *equal keys*. set and map support unique keys. multiset and multimap support equal keys.

For set and multiset the value type is the same as the key type. For map and multimap it is equal to pair<const Key, T>.

iterator of an associative container is of the bidirectional iterator category. insert does not affect the validity of iterators and references to the container, and erase invalidates only the iterators and references to the erased elements.

In the following table, X is an associative container class, a is a value of X, a_uniq is a value of X when X supports unique keys, and a_eq is a value of X when X supports multiple keys, i and j satisfy input iterator requirements and refer to elements of value_type, [i, j) is a valid range, p is a valid iterator to a, q is a dereferenceable iterator to a, [q1, q2) is a valid range in a, t is a value of X::value_type and k is a value of X::key_type.

Table 12: Associative container requirements (in addition to container)

expression	return type	assertion/note pre/post-condition	complexity
`X::key_type`	`Key`		compile time
`X::key_compare`	`Compare`	defaults to `less<key_type>`.	compile time
`X::value_compare`	a binary predicate type	is the same as `key_compare` for `set` and `multiset`; is an ordering relation on pairs induced by the first component (i.e. `Key`) for `map` and `multimap`.	compile time
`X(c)` `X a(c);`		constructs an empty container; uses `c` as a comparison object.	constant
`X()` `X a;`		constructs an empty container; uses `Compare()` as a comparison object.	constant
`X(i, j, c)` `X a(i, j, c);`		constructs an empty container and inserts elements from the range `[i, j)` into it; uses `c` as a comparison object.	`NlogN` in general (`N` is the distance from `i` to `j`); linear if `[i, j)` is sorted with `value_comp()`
`X(i, j)` `X a(i, j);`		same as above, but uses `Compare()` as a comparison object.	same as above
`a.key_comp()`	`X::key_compare`	returns the comparison object out of which `a` was constructed.	constant
`a.value_comp()`	`X::value_compare`	returns an object of `value_compare` constructed out of the comparison object.	constant
`a_uniq.insert(t)`	`pair<iterator, bool>`	inserts `t` if and only if there is no element in the container with key equal to the key of `t`. The `bool` component of the returned pair indicates whether the insertion takes place and the `iterator` component of the pair points to the element with key equal to the key of `t`.	logarithmic
`a_eq.insert(t)`	`iterator`	inserts `t` and returns the iterator pointing to the newly inserted element.	logarithmic

Table 12: Associative container requirements (in addition to container)

expression	return type	assertion/note pre/post-condition	complexity
`a.insert(p, t)`	`iterator`	inserts `t` if and only if there is no element with key equal to the key of `t` in containers with unique keys; always inserts `t` in containers with equal keys. always returns the iterator pointing to the element with key equal to the key of `t`. iterator `p` is a hint pointing to where the insert should start to search.	logarithmic in general, but amortized constant if `t` is inserted right before `p`.
`a.insert(i, j)`	result is not used	inserts the elements from the range `[i, j)` into the container.	`Nlog(size()+N)` (`N` is the distance from `i` to `j`) in general; linear if `[i, j)` is sorted according to `value_comp()`
`a.erase(k)`	`size_type`	erases all the elements in the container with key equal to `k`. returns the number of erased elements.	`log(size()) + count(k)`
`a.erase(q)`	result is not used	erases the element pointed to by `q`.	amortized constant
`a.erase(q1, q2)`	result is not used	erases all the elements in the range `[q1, q2)`.	`log(size())+ N` where `N` is the distance from `q1` to `q2`.
`a.find(k)`	`iterator; const_iterator` for constant `a`	returns an iterator pointing to an element with the key equal to `k`, or `a.end()` if such an element is not found.	logarithmic
`a.count(k)`	`size_type`	returns the number of elements with key equal to `k`.	`log(size())+ count(k)`
`a.lower_bound(k)`	`iterator; const_iterator` for constant `a`	returns an iterator pointing to the first element with key not less than `k`.	logarithmic
`a.upper_bound(k)`	iterator; `const_iterator` for constant `a`	returns an iterator pointing to the first element with key greater than `k`.	logarithmic
`a.equal_range(k)`	`pair<iterator, iterator>; pair<const_iterator, const_iterator>` for constant `a`	equivalent to `make_pair(a.lower_bound(k), a.upper_bound(k))`.	logarithmic

The fundamental property of iterators of associative containers is that they iterate through the containers in the non-descending order of keys where non-descending is defined by the comparison that was used to construct them. For any two dereferenceable iterators `i` and `j` such that distance from `i` to `j` is positive,

```
value_comp(*j, *i) == false
```

For associative containers with unique keys the stronger condition holds,

```
value_comp(*i, *j) == true.
```

8.2.1 Set

`set` is a kind of associative container that supports unique keys (contains at most one of each key value) and provides for fast retrieval of the keys themselves.

```
template <class Key, class Compare = less<Key>,
             template <class U> class Allocator = allocator>
class set {
public:

// typedefs:

      typedef Key key_type;
      typedef Key value_type;
      typedef Allocator<Key>::pointer pointer;
      typedef Allocator<Key>::reference reference;
      typedef Allocator<Key>::const_reference const_reference;
      typedef Compare key_compare;
      typedef Compare value_compare;
      typedef iterator;
      typedef iterator const_iterator;
      typedef size_type;
      typedef difference_type;
      typedef reverse_iterator;
      typedef const_reverse_iterator;

// allocation/deallocation:

      set(const Compare& comp = Compare());
      template <class InputIterator>
      set(InputIterator first, InputIterator last,
             const Compare& comp = Compare());
      set(const set<Key, Compare, Allocator>& x);
      ~set();
      set<Key, Compare, Allocator>& operator=(const set<Key, Compare,
             Allocator>& x);
      void swap(set<Key, Compare, Allocator>& x);

// accessors:

      key_compare key_comp() const;
      value_compare value_comp() const;
      iterator begin() const;
      iterator end() const;
      reverse_iterator rbegin() const;
      reverse_iterator rend() const;
      bool empty() const;
      size_type size() const;
      size_type max_size() const;

// insert/erase:
```

```
        pair<iterator, bool> insert(const value_type& x);
        iterator insert(iterator position, const value_type& x);
        template <class InputIterator>
        void insert(InputIterator first, InputIterator last);
        void erase(iterator position);
        size_type erase(const key_type& x);
        void erase(iterator first, iterator last);

// set operations:

        iterator find(const key_type& x) const;
        size_type count(const key_type& x) const;
        iterator lower_bound(const key_type& x) const;
        iterator upper_bound(const key_type& x) const;
        pair<iterator, iterator> equal_range(const key_type& x) const;
};

template <class Key, class Compare, class Allocator>
bool operator==(const set<Key, Compare, Allocator>& x,
             const set<Key, Compare, Allocator>& y);

template <class Key, class Compare, class Allocator>
bool operator<(const set<Key, Compare, Allocator>& x,
             const set<Key, Compare, Allocator>& y);
```

`iterator` is a constant bidirectional iterator referring to `const value_type`. The exact type is implementation dependent and determined by `Allocator`.

`const_iterator` is the same type as `iterator`.

`size_type` is an unsigned integral type. The exact type is implementation dependent and determined by `Allocator`.

`difference_type` is a signed integral type. The exact type is implementation dependent and determined by `Allocator`.

8.2.2 Multiset

`multiset` is a kind of associative container that supports equal keys (possibly contains multiple copies of the same key value) and provides for fast retrieval of the keys themselves.

```
template <class Key, class Compare = less<Key>,
             template <class U> class Allocator = allocator>
class multiset {
public:

// typedefs:

        typedef Key key_type;
        typedef Key value_type;
        typedef Allocator<Key>::pointer pointer;
        typedef Allocator<Key>::reference reference;
        typedef Allocator<Key>::const_reference const_reference;
        typedef Compare key_compare;
        typedef Compare value_compare;
        typedef iterator;
        typedef iterator const_iterator;
        typedef size_type;
        typedef difference_type;
        typedef reverse_iterator;
```

```
        typedef const_reverse_iterator;

    // allocation/deallocation:

        multiset(const Compare& comp = Compare());
        template <class InputIterator>
        multiset(InputIterator first, InputIterator last,
               const Compare& comp = Compare());
        multiset(const multiset<Key, Compare, Allocator>& x);
        ~multiset();
        multiset<Key, Compare, Allocator>& operator=(const multiset<Key, Compare,
               Allocator>& x);
        void swap(multiset<Key, Compare, Allocator>& x);

    // accessors:

        key_compare key_comp() const;
        value_compare value_comp() const;
        iterator begin() const;
        iterator end() const;
        reverse_iterator rbegin();
        reverse_iterator rend();
        bool empty() const;
        size_type size() const;
        size_type max_size() const;

    // insert/erase:

        iterator insert(const value_type& x);
        iterator insert(iterator position, const value_type& x);
        template <class InputIterator>
        void insert(InputIterator first, InputIterator last);
        void erase(iterator position);
        size_type erase(const key_type& x);
        void erase(iterator first, iterator last);

    // multiset operations:

        iterator find(const key_type& x) const;
        size_type count(const key_type& x) const;
        iterator lower_bound(const key_type& x) const;
        iterator upper_bound(const key_type& x) const;
        pair<iterator, iterator> equal_range(const key_type& x) const;
    };

    template <class Key, class Compare, class Allocator>
    bool operator==(const multiset<Key, Compare, Allocator>& x,
                 const multiset<Key, Compare, Allocator>& y);

    template <class Key, class Compare, class Allocator>
    bool operator<(const multiset<Key, Compare, Allocator>& x,
                 const multiset<Key, Compare, Allocator>& y);
```

`iterator` is a constant bidirectional iterator referring to `const value_type`. The exact type is implementation dependent and determined by `Allocator`.

`const_iterator` is the same type as `iterator`.

`size_type` is an unsigned integral type. The exact type is implementation dependent and determined by `Allocator`.

`difference_type` is a signed integral type. The exact type is implementation dependent and determined by `Allocator`.

8.2.3 Map

`map` is a kind of associative container that supports unique keys (contains at most one of each key value) and provides for fast retrieval of values of another type `T` based on the keys.

```
template <class Key, class T, class Compare = less<Key>,
              template <class U> class Allocator = allocator>
class map {
public:

// typedefs:

       typedef Key key_type;
       typedef pair<const Key, T> value_type;
       typedef Compare key_compare;
       class value_compare
              : public binary_function<value_type, value_type, bool> {
       friend class map;
       protected:
              Compare comp;
              value_compare(Compare c) : comp(c) {}
       public:
              bool operator()(const value_type& x, const value_type& y) {
                     return comp(x.first, y.first);
              }
       };
       typedef iterator;
       typedef const_iterator;
       typedef Allocator<value_type>::pointer pointer;
       typedef Allocator<value_type>::reference reference;
       typedef Allocator<value_type>::const_reference const_reference;
       typedef size_type;
       typedef difference_type;
       typedef reverse_iterator;
       typedef const_reverse_iterator;

// allocation/deallocation:

       map(const Compare& comp = Compare());
       template <class InputIterator>
       map(InputIterator first, InputIterator last,
              const Compare& comp = Compare());
       map(const map<Key, T, Compare, Allocator>& x);
       ~map();
       map<Key, T, Compare, Allocator>&
              operator=(const map<Key, T, Compare, Allocator>& x);
       void swap(map<Key, T, Compare, Allocator>& x);

// accessors:

       key_compare key_comp() const;
       value_compare value_comp() const;
       iterator begin();
```

```
        const_iterator begin() const;
        iterator end();
        const_iterator end() const;
        reverse_iterator rbegin();
        const_reverse_iterator rbegin();
        reverse_iterator rend();
        const_reverse_iterator rend();
        bool empty() const;
        size_type size() const;
        size_type max_size() const;
        Allocator<T>::reference operator[](const key_type& x);

// insert/erase:

        pair<iterator, bool> insert(const value_type& x);
        iterator insert(iterator position, const value_type& x);
        template <class InputIterator>
        void insert(InputIterator first, InputIterator last);
        void erase(iterator position);
        size_type erase(const key_type& x);
        void erase(iterator first, iterator last);

// map operations:

        iterator find(const key_type& x);
        const_iterator find(const key_type& x) const;
        size_type count(const key_type& x) const;
        iterator lower_bound(const key_type& x);
        const_iterator lower_bound(const key_type& x) const;
        iterator upper_bound(const key_type& x);
        const_iterator upper_bound(const key_type& x) const;
        pair<iterator, iterator> equal_range(const key_type& x);
        pair<const_iterator, const_iterator> equal_range(const key_type& x) const;
};

template <class Key, class T, class Compare, class Allocator>
bool operator==(const map<Key, T, Compare, Allocator>& x,
             const map<Key, T, Compare, Allocator>& y);

template <class Key, class T, class Compare, class Allocator>
bool operator<(const map<Key, T, Compare, Allocator>& x,
             const map<Key, T, Compare, Allocator>& y);
```

`iterator` is a bidirectional iterator referring to `value_type`. The exact type is implementation dependent and determined by `Allocator`.

`const_iterator` is a constant bidirectional iterator referring to `const value_type`. The exact type is implementation dependent and determined by `Allocator`. It is guaranteed that there is a constructor for `const_iterator` out of `iterator`.

`size_type` is an unsigned integral type. The exact type is implementation dependent and determined by `Allocator`.

`difference_type` is a signed integral type. The exact type is implementation dependent and determined by `Allocator`.

In addition to the standard set of member functions of associative containers, `map` provides `Allocator<T>::reference operator[](const key_type&)`. For a map `m` and key `k`, `m[k]` is semantically equivalent to `(*((m.insert(make_pair(k, T()))).first)).second`.

8.2.4 Multimap

`multimap` is a kind of associative container that supports equal keys (possibly contains multiple copies of the same key value) and provides for fast retrieval of values of another type `T` based on the keys.

```
template <class Key, class T, class Compare = less<Key>,
              template <class U> class Allocator = allocator>
class multimap {
public:

// typedefs:

       typedef Key key_type;
       typedef pair<const Key, T> value_type;
       typedef Compare key_compare;
       class value_compare
              : public binary_function<value_type, value_type, bool> {
       friend class multimap;
       protected:
              Compare comp;
              value_compare(Compare c) : comp(c) {}
       public:
              bool operator()(const value_type& x, const value_type& y) {
                     return comp(x.first, y.first);
              }
       };
       typedef iterator;
       typedef const_iterator;
       typedef Allocator<value_type>::pointer pointer;
       typedef Allocator<value_type>::reference reference;
       typedef Allocator<value_type>::const_reference const_reference;
       typedef size_type;
       typedef difference_type;
       typedef reverse_iterator;
       typedef const_reverse_iterator;

// allocation/deallocation:

       multimap(const Compare& comp = Compare());
       template <class InputIterator>
       multimap(InputIterator first, InputIterator last,
              const Compare& comp = Compare());
       multimap(const multimap<Key, T, Compare, Allocator>& x);
       ~multimap();
       multimap<Key, T, Compare, Allocator>&
              operator=(const multimap<Key, T, Compare, Allocator>& x);
       void swap(multimap<Key, T, Compare, Allocator>& x);

// accessors:

       key_compare key_comp() const;
       value_compare value_comp() const;
       iterator begin();
       const_iterator begin() const;
       iterator end();
       const_iterator end() const;
       reverse_iterator rbegin();
       const_reverse_iterator rbegin();
       reverse_iterator rend();
```

```
        const_reverse_iterator rend();
        bool empty() const;
        size_type size() const;
        size_type max_size() const;

// insert/erase:

        iterator insert(const value_type& x);
        iterator insert(iterator position, const value_type& x);
        template <class InputIterator>
        void insert(InputIterator first, InputIterator last);
        void erase(iterator position);
        size_type erase(const key_type& x);
        void erase(iterator first, iterator last);

// multimap operations:

        iterator find(const key_type& x);
        const_iterator find(const key_type& x) const;
        size_type count(const key_type& x) const;
        iterator lower_bound(const key_type& x);
        const_iterator lower_bound(const key_type& x) const;
        iterator upper_bound(const key_type& x);
        const_iterator upper_bound(const key_type& x) const;
        pair<iterator, iterator> equal_range(const key_type& x);
        pair<const_iterator, const_iterator> equal_range(const key_type& x) const;
};

template <class Key, class T, class Compare, class Allocator>
bool operator==(const multimap<Key, T, Compare, Allocator>& x,
                const multimap<Key, T, Compare, Allocator>& y);

template <class Key, class T, class Compare, class Allocator>
bool operator<(const multimap<Key, T, Compare, Allocator>& x,
                const multimap<Key, T, Compare, Allocator>& y);
```

`iterator` is a bidirectional iterator referring to `value_type`. The exact type is implementation dependent and determined by `Allocator`.

`const_iterator` is the a constant bidirectional iterator referring to `const value_type`. The exact type is implementation dependent and determined by `Allocator`. It is guaranteed that there is a constructor for `const_iterator` out of `iterator`.

`size_type` is an unsigned integral type. The exact type is implementation dependent and determined by `Allocator`.

`difference_type` is a signed integral type. The exact type is implementation dependent and determined by `Allocator`.

9 Stream iterators

To make it possible for algorithmic templates to work directly with input/output streams, appropriate iterator-like template classes are provided. For example,

```
partial_sum_copy(istream_iterator<double>(cin),istream_iterator<double>(),
                 ostream_iterator<double>(cout, "\n"));
```

reads a file containing floating point numbers from `cin`, and prints the partial sums onto `cout`.

9.1 Istream Iterator

`istream_iterator<T>` reads (using `operator>>`) successive elements from the input stream for which it was constructed. After it is constructed, and every time ++ is used, the iterator reads and stores a value of `T`. If the end of stream is reached (`operator void*()` on the stream returns `false`), the iterator becomes equal to the *end-of-stream* iterator value. The constructor with no arguments `istream_iterator()` always constructs an end of stream input iterator object, which is the only legitimate iterator to be used for the end condition. The result of `operator*` on an end of stream is not defined. For any other iterator value a `const T&` is returned. It is impossible to store things into istream iterators. The main peculiarity of the istream iterators is the fact that ++ operators are not equality preserving, that is, `i == j` does not guarantee at all that `++i == ++j`. Every time ++ is used a new value is read.

The practical consequence of this fact is that istream iterators can be used only for one-pass algorithms, which actually makes perfect sense, since for multi-pass algorithms it is always more appropriate to use in-memory data structures. Two end-of-stream iterators are always equal. An end-of-stream iterator is not equal to a non-end-of-stream iterator. Two non-end-of-stream iterators are equal when they are constructed from the same stream.

```
template <class T, class Distance = ptrdiff_t>
class istream_iterator : public input_iterator<T, Distance> {
friend bool operator==(const istream_iterator<T, Distance>& x,
                       const istream_iterator<T, Distance>& y);
public:
        istream_iterator();
        istream_iterator(istream& s);
        istream_iterator(const istream_iterator<T, Distance>& x);
        ~istream_iterator();
        const T& operator*() const;
        istream_iterator<T, Distance>& operator++();
        istream_iterator<T, Distance> operator++(int);
};
template <class T, class Distance>
bool operator==(const istream_iterator<T, Distance>& x,
             const istream_iterator<T, Distance>& y);
```

9.2 Ostream iterator

`ostream_iterator<T>` writes (using `operator<<`) successive elements onto the output stream from which it was constructed. If it was constructed with `char*` as a constructor argument, this string, called a *delimiter string*, is written to the stream after every `T` is written. It is not possible to get a value out of the output iterator. Its only use is as an output iterator in situations like

```
while (first != last) *result++ = *first++;
```

`ostream_iterator` is defined as:

```
template <class T>
class ostream_iterator : public output_iterator {
public:
        ostream_iterator(ostream& s);
        ostream_iterator(ostream& s, const char* delimiter);
        ostream_iterator(const ostream_iterator<T>& x);
        ~ostream_iterator();
        ostream_iterator<T>& operator=(const T& value);
        ostream_iterator<T>& operator*();
        ostream_iterator<T>& operator++();
        ostream_iterator<T>& operator++(int);
};
```

10 Algorithms

All of the algorithms are separated from the particular implementations of data structures and are parameterized by iterator types. Because of this, they can work with user defined data structures, as long as these data structures have iterator types satisfying the assumptions on the algorithms.

Both in-place and copying versions are provided for certain algorithms. The decision whether to include a copying version was usually based on complexity considerations. When the cost of doing the operation dominates the cost of copy, the copying version is not included. For example, `sort_copy` is not included since the cost of sorting is much more significant, and users might as well do `copy` followed by `sort`. When such a version is provided for *`algorithm`* it is called *`algorithm`*`_copy`. Algorithms that take predicates end with the suffix `_if` (which follows the suffix `_copy`).

The `Predicate` class is used whenever an algorithm expects a function object that when applied to the result of dereferencing the corresponding iterator returns a value convertible to `bool`. In other words, if an algorithm takes `Predicate pred` as its argument and `first` as its iterator argument, it should work correctly in the construct `if (pred(*first)){...}`. The function object `pred` is assumed not to apply any non-constant function through the dereferenced iterator.

The `BinaryPredicate` class is used whenever an algorithm expects a function object that when applied to the result of dereferencing two corresponding iterators or to dereferencing an iterator and type `T` when `T` is part of the signature returns a value convertible to `bool`. In other words, if an algorithm takes `BinaryPredicate binary_pred` as its argument and `first1` and `first2` as its iterator arguments, it should work correctly in the construct `if (binary_pred(*first, *first2)){...}`. `BinaryPredicate` always takes the first iterator type as its first argument, that is, in those cases when `T value` is part of the signature, it should work correctly in the context of `if (binary_pred(*first, value)){...}`. It is expected that `binary_pred` will not apply any non-constant function through the dereferenced iterators.

In the description of the algorithms operators + and – are used for some of the iterator categories for which they do not have to be defined. In these cases the semantics of `a+n` is the same as that of `{ X tmp = a; advance(tmp, n); return tmp; }` and that of `a-b` is the same as that of `{ Distance n; distance(a, b, n); return n; }`.

10.1 Non-mutating sequence operations

10.1.1 For each

```
template <class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f);
```

`for_each` applies `f` to the result of dereferencing every iterator in the range `[first, last)` and returns `f`. `f` is assumed not to apply any non-constant function through the dereferenced iterator. `f` is applied exactly `last - first` times. If `f` returns a result, the result is ignored.

10.1.2 Find

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

template <class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last, Predicate pred);
```

`find` returns the first iterator `i` in the range `[first, last)` for which the following corresponding conditions hold: `*i == value`, `pred(*i) == true`. If no such iterator is found, `last` is returned. Exactly `find(first, last, value) - first` applications of the corresponding predicate are done.

10.1.3 Adjacent find

```
template <class ForwardIterator>
```

```
ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last);

template <class ForwardIterator, class BinaryPredicate>
ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last,
             BinaryPredicate binary_pred);
```

adjacent_find returns the first iterator i such that both i and i + 1 are in the range [first, last) for which the following corresponding conditions hold: *i == *(i + 1), binary_pred(*i, *(i + 1)) == true. If no such iterator i is found, last is returned. At most max((last - first) - 1, 0) applications of the corresponding predicate are done.

10.1.4 Count

```
template <class InputIterator, class T, class Size>
void count(InputIterator first, InputIterator last, const T& value, Size& n);

template <class InputIterator, class Predicate, class Size>
void count_if(InputIterator first, InputIterator last, Predicate pred, Size& n);
```

count adds to n the number of iterators i in the range [first, last) for which the following corresponding conditions hold: *i == value, pred(*i) == true. Exactly last - first applications of the corresponding predicate are done.

count must store the result into a reference argument instead of returning the result because the size type cannot be deduced from built-in iterator types such as int*.

10.1.5 Mismatch

```
template <class InputIterator1, class InputIterator2>
pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1,
             InputIterator1 last1, InputIterator2 first2);

template <class InputIterator1, class InputIterator2, class BinaryPredicate>
pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1,
             InputIterator1 last1, InputIterator2 first2,
             BinaryPredicate binary_pred);
```

mismatch returns a pair of iterators i and j such that j == first2 + (i - first1) and i is the first iterator in the range [first1, last1) for which the following corresponding conditions hold: !(*i == *(first2 + (i - first1))), binary_pred(*i, *(first2 + (i - first1))) == false. If such an iterator i is not found, a pair of last1 and first2 + (last1 - first1) is returned. At most last1 - first1 applications of the corresponding predicate are done.

10.1.6 Equal

```
template <class InputIterator1, class InputIterator2>
bool equal(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2);

template <class InputIterator1, class InputIterator2, class BinaryPredicate>
bool equal(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2,
             BinaryPredicate binary_pred);
```

equal returns true if for every iterator i in the range [first1, last1) the following corresponding conditions hold: *i == *(first2 + (i - first1)),binary_pred(*i, *(first2 + (i - first1))) == true. Otherwise, it returns false. At most last1 - first1 applications of the corresponding predicate are done.

10.1.7 Search

```
template <class ForwardIterator1, class ForwardIterator2>
```

```
ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
            ForwardIterator2 first2, ForwardIterator2 last2);

template <class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
            ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate
            binary_pred);
```

`search` finds a subsequence of equal values in a sequence. `search` returns the first iterator `i` in the range `[first1, last1 - (last2 - first2))` such that for any non-negative integer `n` less than `last2 - first2` the following corresponding conditions hold: `*(i + n) == *(first2 + n)`, `binary_pred(*(i + n), *(first2 + n)) == true`. If no such iterator is found, `last1` is returned. At most `(last1 - first1) * (last2 - first2)` applications of the corresponding predicate are done. The quadratic behavior, however, is highly unlikely.

10.2 Mutating sequence operations

10.2.1 Copy

```
template <class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
            OutputIterator result);
```

`copy` copies elements. For each non-negative integer `n < (last - first)`, `*(result + n) = *(first + n)` is performed. `copy` returns `result + (last - first)`. Exactly `last - first` assignments are done. The result of `copy` is undefined if `result` is in the range `[first, last)`.

```
template <class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2 copy_backward(BidirectionalIterator1 first,
            BidirectionalIterator1 last, BidirectionalIterator2 result);
```

`copy_backward` copies elements in the range `[first, last)` into the range `[result - (last - first), result)` starting from `last - 1` and proceeding to `first`. It should be used instead of `copy` when `last` is in the range `[result - (last - first), result)`. For each positive integer `n <= (last - first)`, `*(result - n) = *(last - n)` is performed. `copy_backward` returns `result - (last - first)`. Exactly `last - first` assignments are done. The result of `copy_backward` is undefined if `result` is in the range `[first, last)`.

10.2.2 Swap

```
template <class T>
void swap(T& a, T& b);
```

`swap` exchanges values stored in two locations.

```
template <class ForwardIterator1, class ForwardIterator2>
void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
```

`iter_swap` exchanges values pointed by the two iterators `a` and `b`.

```
template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1,
            ForwardIterator2 first2);
```

For each non-negative integer `n < (last1 - first1)` the `swap` is performed: `swap(*(first1 + n), *(first2 + n))`. `swap_ranges` returns `first2 + (last1 - first1)`. Exactly `last1 - first1` swaps are done. The result of `swap_ranges` is undefined if the two ranges `[first1, last1)` and `[first2, first2 + (last1 - first1))` overlap.

10.2.3 Transform

```
template <class InputIterator, class OutputIterator, class UnaryOperation>
OutputIterator transform(InputIterator first, InputIterator last,
              OutputIterator result, UnaryOperation op);

template <class InputIterator1, class InputIterator2, class OutputIterator,
              class BinaryOperation>
OutputIterator transform(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, OutputIterator result,
              BinaryOperation binary_op);
```

`transform` assigns through every iterator `i` in the range `[result, result + (last1 - first1))` a new corresponding value equal to `op(*(first1 + (i - result))` or `binary_op(*(first1 + (i - result), *(first2 + (i - result)))`. `transform` returns `result + (last1 - first1)`. Exactly `last1 - first1` applications of `op` or `binary_op` are performed. `op` and `binary_op` are expected not to have any side effects. `result` may be equal to `first` in case of unary transform, or to `first1` or `first2` in case of binary transform.

10.2.4 Replace

```
template <class ForwardIterator, class T>
void replace(ForwardIterator first, ForwardIterator last, const T& old_value,
              const T& new_value);

template <class ForwardIterator, class Predicate, class T>
void replace_if(ForwardIterator first, ForwardIterator last, Predicate pred,
              const T& new_value);
```

`replace` substitutes elements referred by the iterator `i` in the range `[first, last)` with `new_value`, when the following corresponding conditions hold: `*i == old_value`, `pred(*i) == true`. Exactly `last - first` applications of the corresponding predicate are done.

```
template <class InputIterator, class OutputIterator, class T>
OutputIterator replace_copy(InputIterator first, InputIterator last,
              OutputIterator result, const T& old_value, const T& new_value);

template <class Iterator, class OutputIterator, class Predicate, class T>
OutputIterator replace_copy_if(Iterator first, Iterator last,
              OutputIterator result, Predicate pred, const T& new_value);
```

`replace_copy` assigns to every iterator `i` in the range `[result, result + (last - first))` either `new_value` or `*(first + (i - result))` depending on whether the following corresponding conditions hold: `*(first + (i - result)) == old_value`, `pred(*(first + (i - result))) == true`. `replace_copy` returns `result + (last - first)`. Exactly `last - first` applications of the corresponding predicate are done.

10.2.5 Fill

```
template <class ForwardIterator, class T>
void fill(ForwardIterator first, ForwardIterator last, const T& value);

template <class OutputIterator, class Size, class T>
OutputIterator fill_n(OutputIterator first, Size n, const T& value);
```

`fill` assigns `value` through all the iterators in the range `[first, last)` or `[first, first + n)`. `fill_n` returns `first + n`. Exactly `last - first` (or `n`) assignments are done.

10.2.6 Generate

```
template <class ForwardIterator, class Generator>
void generate(ForwardIterator first, ForwardIterator last, Generator gen);

template <class OutputIterator, class Size, class Generator>
OutputIterator generate_n(OutputIterator first, Size n, Generator gen);
```

`generate` invokes the function object `gen` and assigns the return value of `gen` through all the iterators in the range `[first, last)` or `[first, first + n)`. `gen` takes no arguments. `generate_n` returns `first + n`. Exactly `last - first` (or `n`) invocations of `gen` and assignments are done.

10.2.7 Remove

```
template <class ForwardIterator, class T>
ForwardIterator remove(ForwardIterator first, ForwardIterator last,
              const T& value);

template <class ForwardIterator, class Predicate>
ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
              Predicate pred);
```

`remove` eliminates all the elements referred to by iterator `i` in the range `[first, last)` for which the following corresponding conditions hold: `*i == value`, `pred(*i) == true`. `remove` returns the end of the resulting range. `remove` is stable, that is, the relative order of the elements that are not removed is the same as their relative order in the original range. Exactly `last - first` applications of the corresponding predicate are done.

```
template <class InputIterator, class OutputIterator, class T>
OutputIterator remove_copy(InputIterator first, InputIterator last,
              OutputIterator result, const T& value);

template <class InputIterator, class OutputIterator, class Predicate>
OutputIterator remove_copy_if(InputIterator first, InputIterator last,
              OutputIterator result, Predicate pred);
```

`remove_copy` copies all the elements referred to by the iterator `i` in the range `[first, last)` for which the following corresponding conditions do not hold: `*i == value`, `pred(*i) == true`. `remove_copy` returns the end of the resulting range. `remove_copy` is stable, that is, the relative order of the elements in the resulting range is the same as their relative order in the original range. Exactly `last - first` applications of the corresponding predicate are done.

10.2.8 Unique

```
template <class ForwardIterator>
ForwardIterator unique(ForwardIterator first, ForwardIterator last);

template <class ForwardIterator, class BinaryPredicate>
ForwardIterator unique(ForwardIterator first, ForwardIterator last,
              BinaryPredicate binary_pred);
```

`unique` eliminates all but the first element from every consecutive group of equal elements referred to by the iterator `i` in the range `[first, last)` for which the following corresponding conditions hold: `*i == *(i - 1)` or `binary_pred(*i, *(i - 1)) == true`. `unique` returns the end of the resulting range. Exactly `(last - first) - 1` applications of the corresponding predicate are done.

```
template <class InputIterator, class OutputIterator>
OutputIterator unique_copy(InputIterator first, InputIterator last,
              OutputIterator result);

template <class InputIterator, class OutputIterator, class BinaryPredicate>
```

```
OutputIterator unique_copy(InputIterator first, InputIterator last,
            OutputIterator result, BinaryPredicate binary_pred);
```

`unique_copy` copies only the first element from every consecutive group of equal elements referred to by the iterator `i` in the range `[first, last)` for which the following corresponding conditions hold: `*i == *(i - 1)` or `binary_pred(*i, *(i - 1)) == true`. `unique_copy` returns the end of the resulting range. Exactly `last - first` applications of the corresponding predicate are done.

10.2.9 Reverse

```
template <class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);
```

For each non-negative integer `i <= (last - first)/2`, `reverse` applies swap to all pairs of iterators `first + i, (last - i) - 1`. Exactly `(last - first)/2` swaps are performed.

```
template <class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator first,
            BidirectionalIterator last, OutputIterator result);
```

`reverse_copy` copies the range `[first, last)` to the range `[result, result + (last - first))` such that for any non-negative integer `i < (last - first)` the following assignment takes place: `*(result + (last - first) - i) = *(first + i)`. `reverse_copy` returns `result + (last - first)`.Exactly `last - first` assignments are done. The result of `reverse_copy` is undefined if `[first, last)` and `[result, result + (last - first))` overlap.

10.2.10 Rotate

```
template <class ForwardIterator>
void rotate(ForwardIterator first, ForwardIterator middle, ForwardIterator last);
```

For each non-negative integer `i < (last - first)`, `rotate` places the element from the position `first + i` into position `first + (i + (last - middle)) % (last - first)`. `[first, middle)` and `[middle, last)` are valid ranges. At most `last - first` swaps are done.

```
template <class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle,
            ForwardIterator last, OutputIterator result);
```

`rotate_copy` copies the range `[first, last)` to the range `[result, result + (last - first))` such that for each non-negative integer `i < (last - first)` the following assignment takes place: `*(result + (i + (last - middle)) % (last - first)) = *(first + i)`. `rotate_copy` returns `result + (last - first)`. Exactly `last - first` assignments are done. The result of `rotate_copy` is undefined if `[first, last)` and `[result, result + (last - first))` overlap.

10.2.11 Random shuffle

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

template <class RandomAccessIterator, class RandomNumberGenerator>
void random_shuffle(RandomAccessIterator first, RandomAccessIterator last,
            RandomNumberGenerator& rand);
```

`random_shuffle` shuffles the elements in the range `[first, last)` with uniform distribution. Exactly `(last - first) - 1` swaps are done. `random_shuffle` can take a particular random number generating function object `rand` such that `rand` takes a positive argument `n` of distance type of the `RandomAccessIterator` and returns a randomly chosen value between `0` and `n-1`.

10.2.12 Partitions

```
template <class BidirectionalIterator, class Predicate>
BidirectionalIterator partition(BidirectionalIterator first,
              BidirectionalIterator last, Predicate pred);
```

`partition` places all the elements in the range `[first, last)` that satisfy `pred` before all the elements that do not satisfy it. It returns an iterator `i` such that for any iterator `j` in the range `[first, i)`, `pred(*j) == true`, and for any iterator `k` in the range `[i, last)`,`pred(*j) == false`. It does at most `(last - first)/2` swaps. Exactly `last - first` applications of the predicate is done.

```
template <class BidirectionalIterator, class Predicate>
BidirectionalIterator stable_partition(BidirectionalIterator first,
              BidirectionalIterator last, Predicate pred);
```

`stable_partition` places all the elements in the range `[first, last)` that satisfy `pred` before all the elements that do not satisfy it. It returns an iterator `i` such that for any iterator `j` in the range `[first, i)`, `pred(*j) == true`, and for any iterator `k` in the range `[i, last)`,`pred(*j) == false`. The relative order of the elements in both groups is preserved. It does at most `(last - first) * log(last - first)` swaps, but only linear number of swaps if there is enough extra memory. Exactly `last - first` applications of the predicate are done.

10.3 Sorting and related operations

All the operations in this section have two versions: one that takes a function object of type `Compare` and one that uses an `operator<`.

`Compare` is a function object which returns a value convertible to `bool`. `Compare comp` is used throughout for algorithms assuming an ordering relation. `comp` satisfies the standard axioms for total ordering and it does not apply any non-constant function through the dereferenced iterator. For all algorithms that take `Compare`, there is a version that uses `operator<` instead. That is, `comp(*i, *j) == true` defaults to `*i < *j == true`.

A sequence is sorted with respect to a comparator `comp` if for any iterator `i` pointing to an element in a sequence and any non-negative integer `n` such that `i + n` is a valid iterator pointing to an element of the same sequence, `comp(*(i + n), *i) == false`.

In the descriptions of the functions that deal with ordering relationships we frequently use a notion of equality to describe concepts such as stability. The equality to which we refer is not necessarily an `operator==`, but an equality relation induced by the total ordering. That is, two element `a` and `b` are considered equal if and only if `!(a < b) && !(b < a)`.

10.3.1 Sort

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

template <class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

`sort` sorts the elements in the range `[first, last)`. It does approximately `NlogN` (where `N` equals to `last - first`) comparisons on the average. If the worst case behavior is important `stable_sort` or `partial_sort` should be used.

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

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

```
                    Compare comp);
```

`stable_sort` sorts the elements in the range `[first, last)`. It is stable, that is, the relative order of the equal elements is preserved. It does at most `N(logN)`^{2} (where `N` equals to `last - first`) comparisons; if enough extra memory is available, it is `NlogN`.

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

template <class RandomAccessIterator, class Compare>
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                  RandomAccessIterator last, Compare comp);
```

`partial_sort` places the first `middle - first` sorted elements from the range `[first, last)` into the range `[first, middle)`. The rest of the elements in the range `[middle, last)` are placed in an undefined order. It takes approximately `(last - first) * log(middle - first)` comparisons.

```
template <class InputIterator, class RandomAccessIterator>
RandomAccessIterator partial_sort_copy(InputIterator first, InputIterator last,
                  RandomAccessIterator result_first, RandomAccessIterator
                  result_last);

template <class InputIterator, class RandomAccessIterator, class Compare>
RandomAccessIterator partial_sort_copy(InputIterator first, InputIterator last,
                  RandomAccessIterator result_first, RandomAccessIterator
                  result_last, Compare comp);
```

`partial_sort_copy` places the first `min(last - first, result_last - result_first)` sorted elements into the range `[result_first, result_first + min(last - first, result_last - result_first))`. It returns either `result_last` or `result_first + (last - first)` whichever is smaller. It takes approximately `(last - first) * log(min(last - first, result_last - result_first))` comparisons.

10.3.2 Nth element

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

template <class RandomAccessIterator, class Compare>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                  RandomAccessIterator last, Compare comp);
```

After `nth_element` the element in the position pointed to by `nth` is the element that would be in that position if the whole range were sorted. Also for any iterator `i` in the range `[first, nth)` and any iterator `j` in the range `[nth, last)` it holds that `!(*i > *j)` or `comp(*i, *j) == false`. It is linear on the average.

10.3.3 Binary search

All of the algorithms in this section are versions of binary search. They work on non-random access iterators minimizing the number of comparisons, which will be logarithmic for all types of iterators. They are especially appropriate for random access iterators, since these algorithms do a logarithmic number of steps through the data structure. For non-random access iterators they execute a linear number of steps.

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

```
template <class ForwardIterator, class T, class Compare>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
             const T& value, Compare comp);
```

`lower_bound` finds the first position into which `value` can be inserted without violating the ordering. `lower_bound` returns the furthermost iterator `i` in the range `[first, last)` such that for any iterator `j` in the range `[first, i)` the following corresponding conditions hold: `*j < value` or `comp(*j, value) == true`. At most `log(last - first) + 1` comparisons are done.

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

template <class ForwardIterator, class T, class Compare>
ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
             const T& value, Compare comp);
```

`upper_bound` finds the furthermost position into which `value` can be inserted without violating the ordering. `upper_bound` returns the furthermost iterator `i` in the range `[first, last)` such that for any iterator `j` in the range `[first, i)` the following corresponding conditions hold: `!(value < *j)` or `comp(value, *j) == false`. At most `log(last - first) + 1` comparisons are done.

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

template <class ForwardIterator, class T, class Compare>
pair<ForwardIterator, ForwardIterator> equal_range(ForwardIterator first,
             ForwardIterator last, const T& value, Compare comp);
```

`equal_range` finds the largest subrange `[i, j)` such that the value can be inserted at any iterator `k` in it. `k` satisfies the corresponding conditions: `!(*k < value) && !(value < *k)` or `comp(*k, value) == false && comp(value, *k) == false`. At most `2 * log(last - first) + 1` comparisons are done.

```
template <class ForwardIterator, class T>
bool binary_search(ForwardIterator first, ForwardIterator last, const T& value);

template <class ForwardIterator, class T, class Compare>
bool binary_search(ForwardIterator first, ForwardIterator last, const T& value,
             Compare comp);
```

`binary_search` returns `true` if there is an iterator `i` in the range `[first last)` that satisfies the corresponding conditions: `!(*i < value) && !(value < *i)` or `comp(*i, value) == false && comp(value, *i) == false`. At most `log(last - first) + 2` comparisons are done.

10.3.4 Merge

```
template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2,
             OutputIterator result);

template <class InputIterator1, class InputIterator2, class OutputIterator,
       class Compare>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2,
             OutputIterator result, Compare comp);
```

`merge` merges two sorted ranges `[first1, last1)` and `[first2, last2)` into the range `[result, result + (last1 - first1) + (last2 - first2))`. The merge is *stable*, that is, for equal elements in the two ranges, the elements from the first range always precede the elements from the second. `merge` returns `result + (last1 - first1) + (last2 - first2)`. At most `(last1 - first1) + (last2 - first2) - 1` comparisons are performed. The result of `merge` is undefined if the resulting range overlaps with either of the original ranges.

```
template <class BidirectionalIterator>
void inplace_merge(BidirectionalIterator first, BidirectionalIterator middle,
             BidirectionalIterator last);

template <class BidirectionalIterator, class Compare>
void inplace_merge(BidirectionalIterator first, BidirectionalIterator middle,
             BidirectionalIterator last, Compare comp);
```

`inplace_merge` merges two sorted consecutive ranges `[first, middle)` and `[middle, last)` putting the result of the merge into the range `[first, last)`. The merge is *stable*, that is, for equal elements in the two ranges, the elements from the first range always precede the elements from the second. When enough additional memory is available, at most `(last - first) - 1` comparisons are performed. If no additional memory is available, an algorithm with `O(NlogN)` complexity may be used.

10.3.5 Set operations on sorted structures

This section defines all the basic set operations on sorted structures. They even work with multisets containing multiple copies of equal elements. The semantics of the set operations is generalized to multisets in a standard way by defining union to contain the maximum number of occurrences of every element, intersection to contain the minimum, and so on.

```
template <class InputIterator1, class InputIterator2>
bool includes(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2);

template <class InputIterator1, class InputIterator2, class Compare>
bool includes(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2, Compare comp);
```

`includes` returns `true` if every element in the range `[first2, last2)` is contained in the range `[first1, last1)`. It returns `false` otherwise. At most `((last1 - first1) + (last2 - first2)) * 2 - 1` comparisons are performed.

```
template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2,
             OutputIterator result);

template <class InputIterator1, class InputIterator2, class OutputIterator, class
             Compare>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2,
             OutputIterator result, Compare comp);
```

`set_union` constructs a sorted union of the elements from the two ranges. It returns the end of the constructed range. `set_union` is stable, that is, if an element is present in both ranges, the one from the first range is copied. At most `((last1 - first1) + (last2 - first2)) * 2 - 1` comparisons are performed. The result of `set_union` is undefined if the resulting range overlaps with either of the original ranges.

```
template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1,
```

```
                InputIterator2 first2, InputIterator2 last2,
                OutputIterator result);

template <class InputIterator1, class InputIterator2, class OutputIterator,
      class Compare>
OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2, OutputIterator result,
                Compare comp);
```

`set_intersection` constructs a sorted intersection of the elements from the two ranges. It returns the end of the constructed range. `set_intersection` is guaranteed to be stable, that is, if an element is present in both ranges, the one from the first range is copied. At most `((last1 - first1) + (last2 - first2)) * 2 - 1` comparisons are performed. The result of `set_intersection` is undefined if the resulting range overlaps with either of the original ranges.

```
template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2,
                OutputIterator result);

template <class InputIterator1, class InputIterator2, class OutputIterator,
      class Compare>
OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2,
                OutputIterator result, Compare comp);
```

`set_difference` constructs a sorted difference of the elements from the two ranges. It returns the end of the constructed range. At most `((last1 - first1) + (last2 - first2)) * 2 - 1` comparisons are performed. The result of `set_difference` is undefined if the resulting range overlaps with either of the original ranges.

```
template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_symmetric_difference(InputIterator1 first1, InputIterator1
                last1, InputIterator2 first2, InputIterator2 last2,
                OutputIterator result);

template <class InputIterator1, class InputIterator2, class OutputIterator,
      class Compare>
OutputIterator set_symmetric_difference(InputIterator1 first1, InputIterator1
                last1, InputIterator2 first2, InputIterator2 last2,
                OutputIterator result, Compare comp);
```

`set_symmetric_difference` constructs a sorted symmetric difference of the elements from the two ranges. It returns the end of the constructed range. At most `((last1 - first1) + (last2 - first2)) * 2 - 1` comparisons are performed. The result of `set_symmetric_difference` is undefined if the resulting range overlaps with either of the original ranges.

10.3.6 Heap operations

A heap is a particular organization of elements in a range between two random access iterators `[a, b)`. Its two key properties are: (1) `*a` is the largest element in the range and (2) `*a` may be removed by `pop_heap`, or a new element added by `push_heap`, in O(`logN`) time. These properties make heaps useful as priority queues. `make_heap` converts a range into a heap and `sort_heap` turns a heap into a sorted sequence.

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

template <class RandomAccessIterator, class Compare>
void push_heap(RandomAccessIterator first, RandomAccessIterator last,
```

```
                    Compare comp);
```

push_heap assumes the range [first, last - 1) is a valid heap and properly places the value in the location last - 1 into the resulting heap [first, last). At most log(last - first) comparisons are performed.

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

template <class RandomAccessIterator, class Compare>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
              Compare comp);
```

pop_heap assumes the range [first, last) is a valid heap, then swaps the value in the location first with the value in the location last - 1 and makes [first, last - 1) into a heap. At most 2 * log(last - first) comparisons are performed.

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

template <class RandomAccessIterator, class Compare>
void make_heap(RandomAccessIterator first, RandomAccessIterator last,
              Compare comp);
```

make_heap constructs a heap out of the range [first, last). At most 3*(last - first) comparisons are performed.

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

template <class RandomAccessIterator, class Compare>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
              Compare comp);
```

sort_heap sorts elements in the heap [first, last). At most NlogN comparisons are performed where N is equal to last - first. sort_heap is not stable.

10.3.7 Minimum and maximum

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

template <class T, class Compare>
const T& min(const T& a, const T& b, Compare comp);

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

template <class T, class Compare>
const T& max(const T& a, const T& b, Compare comp);
```

min returns the smaller and max the larger. min and max return the first argument when their arguments are equal.

```
template <class ForwardIterator>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last);

template <class ForwardIterator, class Compare>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last,
              Compare comp);
```

`max_element` returns the first iterator `i` in the range `[first, last)` such that for any iterator `j` in the range `[first, last)` the following corresponding conditions hold: `!(*i < *j)` or `comp(*i, *j) == false`. Exactly `max((last - first) - 1, 0)` applications of the corresponding comparisons are done.

```
template <class ForwardIterator>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last);

template <class ForwardIterator, class Compare>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last,
                            Compare comp);
```

`min_element` returns the first iterator `i` in the range `[first, last)` such that for any iterator `j` in the range `[first, last)` the following corresponding conditions hold: `!(*j < *i)` or `comp(*j, *i) == false`. Exactly `max((last - first) - 1, 0)` applications of the corresponding comparisons are done.

10.3.8 Lexicographical comparison

```
template <class InputIterator1, class InputIterator2>
bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2);

template <class InputIterator1, class InputIterator2, class Compare>
bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2, Compare comp);
```

`lexicographical_compare` returns `true` if the sequence of elements defined by the range `[first1, last1)` is lexicographically less than the sequence of elements defined by the range `[first2, last2)`. It returns `false` otherwise. At most `2 * min((last1 - first1), (last2 - first2))` applications of the corresponding comparison are done.

10.3.9 Permutation generators

```
template <class BidirectionalIterator>
bool next_permutation(BidirectionalIterator first, BidirectionalIterator last);

template <class BidirectionalIterator, class Compare>
bool next_permutation(BidirectionalIterator first, BidirectionalIterator last,
                      Compare comp);
```

`next_permutation` takes a sequence defined by the range `[first, last)` and transforms it into the *next* permutation. The next permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to `operator<` or `comp`. If such a permutation exists, it returns `true`. Otherwise, it transforms the sequence into the smallest permutation, that is, the ascendingly sorted one, and returns `false`. At most `(last - first)/2` swaps are performed.

```
template <class BidirectionalIterator>
bool prev_permutation(BidirectionalIterator first, BidirectionalIterator last);

template <class BidirectionalIterator, class Compare>
bool prev_permutation(BidirectionalIterator first, BidirectionalIterator last,
                      Compare comp);
```

`prev_permutation` takes a sequence defined by the range `[first, last)` and transforms it into the *previous* permutation. The previous permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to `operator<` or `comp`. If such a permutation exists, it returns `true`. Otherwise, it transforms the sequence into the largest permutation, that is, the descendingly sorted one, and returns `false`. At most `(last - first)/2` swaps are performed.

10.4 Generalized numeric operations

10.4.1 Accumulate

```
template <class InputIterator, class T>
T accumulate(InputIterator first, InputIterator last, T init);

template <class InputIterator, class T, class BinaryOperation>
T accumulate(InputIterator first, InputIterator last, T init,
             BinaryOperation binary_op);
```

`accumulate` is similar to the APL *reduction* operator and Common Lisp *reduce* function, but it avoids the difficulty of defining the result of reduction on an empty sequence by always requiring an initial value. Accumulation is done by initializing the accumulator `acc` with the initial value `init` and then modifying it with `acc = acc + *i` or `acc = binary_op(acc, *i)` for every iterator `i` in the range `[first, last)` in order. `binary_op` is assumed not to cause side effects.

10.4.2 Inner product

```
template <class InputIterator1, class InputIterator2, class T>
T inner_product(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, T init);

template <class InputIterator1, class InputIterator2, class T,
             class BinaryOperation1, class BinaryOperation2>
T inner_product(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, T init,
             BinaryOperation1 binary_op1, BinaryOperation2 binary_op2);
```

`inner_product` computes its result by initializing the accumulator `acc` with the initial value `init` and then modifying it with `acc = acc + (*i1) * (*i2)` or `acc = binary_op1(acc, binary_op2(*i1, *i2))` for every iterator `i1` in the range `[first, last)` and iterator `i2` in the range `[first2, first2 + (last - first))` in order. `binary_op1` and `binary_op2` are assumed not to cause side effects.

10.4.3 Partial sum

```
template <class InputIterator, class OutputIterator>
OutputIterator partial_sum(InputIterator first, InputIterator last,
             OutputIterator result);

template <class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator partial_sum(InputIterator first, InputIterator last,
             OutputIterator result, BinaryOperation binary_op);
```

`partial_sum` assigns to every iterator `i` in the range `[result, result + (last - first))` a value correspondingly equal to `((...(*first + *(first + 1)) + ... )+ *(first + (i - result)))` or `binary_op(binary_op(..., binary_op(*first, *(first + 1)),...), *(first + (i - result)))`. `partial_sum` returns `result + (last - first)`. Exactly `(last - first) - 1` applications of `binary_op` are performed. `binary_op` is expected not to have any side effects. `result` may be equal to `first`.

10.4.4 Adjacent difference

```
template <class InputIterator, class OutputIterator>
OutputIterator adjacent_difference(InputIterator first, InputIterator last,
             OutputIterator result);

template <class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator adjacent_difference(InputIterator first, InputIterator last,
             OutputIterator result, BinaryOperation binary_op);
```

adjacent_difference assigns to every element referred to by iterator i in the range [result + 1, result + (last - first)) a value correspondingly equal to *(first + (i - result)) - *(first + (i - result) - 1) or binary_op(*(first + (i - result)), *(first + (i - result) - 1)). result gets the value of *first. adjacent_difference returns result + (last - first). Exactly (last - first) - 1 applications of binary_op are performed. binary_op is expected not to have any side effects. result may be equal to first.

11 Adaptors

Adaptors are template classes that provide interface mappings. For example, insert_iterator provides a container with an output iterator interface.

11.1 Container adaptors

It is often useful to provide restricted interfaces to containers. The library provides stack, queue and priority_queue through the adaptors that can work with different sequence types.

11.1.1 Stack

Any sequence supporting operations back, push_back and pop_back can be used to instantiate stack. In particular, vector, list and deque can be used.

```
template <class Container>
class stack {
friend bool operator==(const stack<Container>& x, const stack<Container>& y);
friend bool operator<(const stack<Container>& x, const stack<Container>& y);
public:
	typedef Container::value_type value_type;
	typedef Container::size_type size_type;
protected:
	Container c;
public:
	bool empty() const { return c.empty(); }
	size_type size() const { return c.size(); }
	value_type& top() { return c.back(); }
	const value_type& top() const { return c.back(); }
	void push(const value_type& x) { c.push_back(x); }
	void pop() { c.pop_back(); }
};

template <class Container>
bool operator==(const stack<Container>& x, const stack<Container>& y) {
	return x.c == y.c;
}

template <class Container>
bool operator<(const stack<Container>& x, const stack<Container>& y) {
	return x.c < y.c;
}
```

For example, stack<vector<int> > is an integer stack made out of vector, and stack<deque<char> > is a character stack made out of deque.

11.1.2 Queue

Any sequence supporting operations front, back, push_back and pop_front can be used to instantiate queue. In particular, list and deque can be used.

```
template <class Container>
```

```
class queue {
friend bool operator==(const queue<Container>& x, const queue<Container>& y);
friend bool operator<(const queue<Container>& x, const queue<Container>& y);
public:
	typedef Container::value_type value_type;
	typedef Container::size_type size_type;
protected:
	Container c;
public:
	bool empty() const { return c.empty(); }
	size_type size() const { return c.size(); }
	value_type& front() { return c.front(); }
	const value_type& front() const { return c.front(); }
	value_type& back() { return c.back(); }
	const value_type& back() const { return c.back(); }
	void push(const value_type& x) { c.push_back(x); }
	void pop() { c.pop_front(); }
};

template <class Container>
bool operator==(const queue<Container>& x, const queue<Container>& y) {
	return x.c == y.c;
}

template <class Container>
bool operator<(const queue<Container>& x, const queue<Container>& y) {
	return x.c < y.c;
}
```

11.1.3 Priority queue

Any sequence with random access iterator and supporting operations `front`, `push_back` and `pop_back` can be used to instantiate `priority_queue`. In particular, `vector` and `deque` can be used.

```
template <class Container, class Compare = less<Container::value_type> >
class priority_queue {
public:
	typedef Container::value_type value_type;
	typedef Container::size_type size_type;
protected:
	Container c;
	Compare comp;
public:
	priority_queue(const Compare& x = Compare()) : c(), comp(x) {}
	template <class InputIterator>
	priority_queue(InputIterator first, InputIterator last,
			const Compare& x = Compare()) : c(first, last), comp(x) {
		make_heap(c.begin(), c.end(), comp);
	}
	bool empty() const { return c.empty(); }
	size_type size() const { return c.size(); }
	const value_type& top() const { return c.front(); }
	void push(const value_type& x) {
		c.push_back(x);
		push_heap(c.begin(), c.end(), comp);
	}
	void pop() {
		pop_heap(c.begin(), c.end(), comp);
		c.pop_back();
```

```
        }
};

// no equality is provided
```

11.2 Iterator adaptors

11.2.1 Reverse iterators

Bidirectional and random access iterators have corresponding reverse iterator adaptors that iterate through the data structure in the opposite direction. They have the same signatures as the corresponding iterators. The fundamental relation between a reverse iterator and its corresponding iterator `i` is established by the identity

```
&*(reverse_iterator(i)) == &*(i - 1).
```

This mapping is dictated by the fact that while there is always a pointer past the end of an array, there might not be a valid pointer before the beginning of an array.

```
template <class BidirectionalIterator, class T, class Reference = T&,
        class Distance = ptrdiff_t>
class reverse_bidirectional_iterator
        : public bidirectional_iterator<T, Distance> {
        typedef reverse_bidirectional_iterator<BidirectionalIterator, T,
                                        Reference, Distance> self;
        friend bool operator==(const self& x, const self& y);
protected:
        BidirectionalIterator current;
public:
        reverse_bidirectional_iterator() {}
        reverse_bidirectional_iterator(BidirectionalIterator x) : current(x) {}
        BidirectionalIterator base() { return current; }
        Reference operator*() const {
                BidirectionalIterator tmp = current;
                return *--tmp;
        }
        self& operator++() {
                --current;
                return *this;
        }
        self operator++(int) {
                self tmp = *this;
                --current;
                return tmp;
        }
        self& operator--() {
                ++current;
                return *this;
        }
        self operator--(int) {
                self tmp = *this;
                ++current;
                return tmp;
        }
};

template <class BidirectionalIterator, class T, class Reference, class Distance>
inline bool operator==(
                const reverse_bidirectional_iterator<BidirectionalIterator, T,
                                                Reference, Distance>& x,
```

```
        const reverse_bidirectional_iterator<BidirectionalIterator, T,
                                          Reference, Distance>& y) {
    return x.current == y.current;
}

template <class RandomAccessIterator, class T, class Reference = T&,
    class Distance = ptrdiff_t>
class reverse_iterator : public random_access_iterator<T, Distance> {
    typedef reverse_iterator<RandomAccessIterator, T, Reference, Distance>
        self;
    friend bool operator==(const self& x, const self& y);
    friend bool operator<(const self& x, const self& y);
    friend Distance operator-(const self& x, const self& y);
    friend self operator+(Distance n, const self& x);
protected:
    RandomAccessIterator current;
public:
    reverse_iterator() {}
    reverse_iterator(RandomAccessIterator x) : current(x) {}
    RandomAccessIterator base() { return current; }
    Reference operator*() const {
        RandomAccessIterator tmp = current;
        return *--tmp;
    }
    self& operator++() {
        --current;
        return *this;
    }
    self operator++(int) {
        self tmp = *this;
        --current;
        return tmp;
    }
    self& operator--() {
        ++current;
        return *this;
    }
    self operator--(int) {
        self tmp = *this;
        ++current;
        return tmp;
    }
    self operator+(Distance n) const {
        return self(current - n);
    }
    self&  operator+=(Distance n) {
        current -= n;
        return *this;
    }
    self operator-(Distance n) const {
        return self(current + n);
    }
    self& operator-=(Distance n) {
        current += n;
        return *this;
    }
    Reference operator[](Distance n) { return *(*this + n); }
};
```

```
template <class RandomAccessIterator, class T, class Reference, class Distance>
inline bool operator==(
       const reverse_iterator<RandomAccessIterator, T, Reference, Distance>& x,
       const reverse_iterator<RandomAccessIterator, T, Reference, Distance>& y) {
       return x.current == y.current;
}

template <class RandomAccessIterator, class T, class Reference, class Distance>
inline bool operator<(
       const reverse_iterator<RandomAccessIterator, T, Reference, Distance>& x,
       const reverse_iterator<RandomAccessIterator, T, Reference, Distance>& y) {
       return y.current < x.current;
}

template <class RandomAccessIterator, class T, class Reference, class Distance>
inline Distance operator-(
       const reverse_iterator<RandomAccessIterator, T, Reference, Distance>& x,
       const reverse_iterator<RandomAccessIterator, T, Reference, Distance>& y) {
       return y.current - x.current;
}

template <class RandomAccessIterator, class T, class Reference, class Distance>
inline reverse_iterator<RandomAccessIterator, T, Reference, Distance> operator+(
       Distance n,
       const reverse_iterator<RandomAccessIterator, T, Reference, Distance>& x) {
       return reverse_iterator<RandomAccessIterator, T, Reference, Distance>
              (x.current - n);
}
```

11.2.2 Insert iterators

To make it possible to deal with insertion in the same way as writing into an array, a special kind of iterator adaptors, called *insert iterators*, are provided in the library. With regular iterator classes,

```
while (first != last) *result++ = *first++;
```

causes a range `[first, last)` to be copied into a range starting with `result`. The same code with `result` being an insert iterator will insert corresponding elements into the container. This device allows all of the copying algorithms in the library to work in the *insert mod*e instead of the regular overwrite mode.

An insert iterator is constructed from a container and possibly one of its iterators pointing to where insertion takes place if it is neither at the beginning nor at the end of the container. Insert iterators satisfy the requirements of output iterators. `operator*` returns the insert iterator itself. The assignment `operator=(const T& x)` is defined on insert iterators to allow writing into them, it inserts `x` right before where the insert iterator is pointing. In other words, an insert iterator is like a cursor pointing into the container where the insertion takes place. `back_insert_iterator` inserts elements at the end of a container, `front_insert_iterator` inserts elements at the beginning of a container, and `insert_iterator` inserts elements where the iterator points to in a container. `back_inserter`, `front_inserter`, and `inserter` are three functions making the insert iterators out of a container.

```
template <class Container>
class back_insert_iterator : public output_iterator {
protected:
       Container& container;
public:
       back_insert_iterator(Container& x) : container(x) {}
       back_insert_iterator<Container>&
       operator=(const Container::value_type& value) {
              container.push_back(value);
```

```
		return *this;
	}
	back_insert_iterator<Container>& operator*() { return *this; }
	back_insert_iterator<Container>& operator++() { return *this; }
	back_insert_iterator<Container>& operator++(int) { return *this; }
};

template <class Container>
back_insert_iterator<Container> back_inserter(Container& x) {
	return back_insert_iterator<Container>(x);
}

template <class Container>
class front_insert_iterator : public output_iterator {
protected:
	Container& container;
public:
	front_insert_iterator(Container& x) : container(x) {}
	front_insert_iterator<Container>&
	operator=(const Container::value_type& value) {
		container.push_front(value);
		return *this;
	}
	front_insert_iterator<Container>& operator*() { return *this; }
	front_insert_iterator<Container>& operator++() { return *this; }
	front_insert_iterator<Container>& operator++(int) { return *this; }
};

template <class Container>
front_insert_iterator<Container> front_inserter(Container& x) {
	return front_insert_iterator<Container>(x);
}

template <class Container>
class insert_iterator : public output_iterator {
protected:
	Container& container;
	Container::iterator iter;
public:
	insert_iterator(Container& x, Container::iterator i)
		: container(x), iter(i) {}
	insert_iterator<Container>& operator=(const Container::value_type& value) {
		iter = container.insert(iter, value);
		++iter;
		return *this;
	}
	insert_iterator<Container>& operator*() { return *this; }
	insert_iterator<Container>& operator++() { return *this; }
	insert_iterator<Container>& operator++(int) { return *this; }
};

template <class Container, class Iterator>
insert_iterator<Container> inserter(Container& x, Iterator i) {
	return insert_iterator<Container>(x, Container::iterator(i));
}
```

11.3 Function adaptors

Function adaptors work only with function object classes with argument types and result type defined.

11.3.1 Negators

Negators `not1` and `not2` take a unary and a binary predicate correspondingly and return their complements.

```
template <class Predicate>
class unary_negate : public unary_function<Predicate::argument_type, bool> {
protected:
       Predicate pred;
public:
       unary_negate(const Predicate& x) : pred(x) {}
       bool operator()(const argument_type& x) const { return !pred(x); }
};

template <class Predicate>
unary_negate<Predicate> not1(const Predicate& pred) {
       return unary_negate<Predicate>(pred);
}

template <class Predicate>
class binary_negate : public binary_function<Predicate::first_argument_type,
                                         Predicate::second_argument_type, bool> {
protected:
       Predicate pred;
public:
       binary_negate(const Predicate& x) : pred(x) {}
       bool operator()(const first_argument_type& x,
                      const second_argument_type& y) const {
              return !pred(x, y);
       }
};

template <class Predicate>
binary_negate<Predicate> not2(const Predicate& pred) {
       return binary_negate<Predicate>(pred);
}
```

11.3.2 Binders

Binders `bind1st` and `bind2nd` take a function object `f` of two arguments and a value `x` and return a function object of one argument constructed out of `f` with the first or second argument correspondingly bound to `x`.

```
template <class Operation>
class binder1st : public unary_function<Operation::second_argument_type,
                                         Operation::result_type> {
protected:
       Operation op;
       Operation::first_argument_type value;
public:
       binder1st(const Operation& x, const Operation::first_argument_type& y)
              : op(x), value(y) {}
       result_type operator()(const argument_type& x) const {
              return op(value, x);
       }
};

template <class Operation, class T>
binder1st<Operation> bind1st(const Operation& op, const T& x) {
       return binder1st<Operation>(op, Operation::first_argument_type(x));
```

```
}

template <class Operation>
class binder2nd : public unary_function<Operation::first_argument_type,
                                        Operation::result_type> {
protected:
	Operation op;
	Operation::second_argument_type value;
public:
	binder2nd(const Operation& x, const Operation::second_argument_type& y)
		: op(x), value(y) {}
	result_type operator()(const argument_type& x) const {
		return op(x, value);
	}
};

template <class Operation, class T>
binder2nd<Operation> bind2nd(const Operation& op, const T& x) {
	return binder2nd<Operation>(op, Operation::second_argument_type(x));
}
```

For example, `find_if(v.begin(), v.end(), bind2nd(greater<int>(), 5))` finds the first integer in vector `v` greater than 5; `find_if(v.begin(), v.end(), bind1st(greater<int>(), 5))` finds the first integer in `v` less than 5.

11.3.3 Adaptors for pointers to functions

To allow pointers to (unary and binary) functions to work with function adaptors the library provides:

```
template <class Arg, class Result>
class pointer_to_unary_function : public unary_function<Arg, Result> {
protected:
	Result (*ptr)(Arg);
public:
	pointer_to_unary_function() {}
	pointer_to_unary_function(Result (*x)(Arg)) : ptr(x) {}
	Result operator()(Arg x) const { return ptr(x); }
};

template <class Arg, class Result>
pointer_to_unary_function<Arg, Result> ptr_fun(Result (*x)(Arg)) {
	return pointer_to_unary_function<Arg, Result>(x);
}

template <class Arg1, class Arg2, class Result>
class pointer_to_binary_function : public binary_function<Arg1, Arg2, Result> {
protected:
	Result (*ptr)(Arg1, Arg2);
public:
	pointer_to_binary_function() {}
	pointer_to_binary_function(Result (*x)(Arg1, Arg2)) : ptr(x) {}
	Result operator()(Arg1 x, Arg2 y) const { return ptr(x, y); }
};

template <class Arg1, class Arg2, class Result>
pointer_to_binary_function<Arg1, Arg2, Result>
ptr_fun(Result (*x)(Arg1, Arg2)) {
	return pointer_to_binary_function<Arg1, Arg2, Result>(x);
}
```

For example, `replace_if(v.begin(), v.end(), not1(bind2nd(ptr_fun(strcmp), "C")), "C++")` replaces all the `"C"` with `"C++"` in sequence `v`.

Compilation systems that have multiple pointer to function types have to provide additional `ptr_fun` template functions.

12 Memory Handling Primitives

To obtain a typed pointer to an uninitialized memory buffer of a given size the following function is defined:

```
template <class T>
inline T* allocate(ptrdiff_t n, T*); // n >= 0
```

The size (in bytes) of the allocated buffer is no less than `n*sizeof(T)`.

For every memory model there is a corresponding `allocate` template function defined with the first argument type being the distance type of the pointers in the memory model.

For example, if a compilation system supports `__huge` pointers with the distance type being `long long`, the following template function is provided:

```
template <class T>
inline T __huge* allocate(long long n, T __huge *);
```

Also, the following functions are provided:

```
template <class T>
inline void deallocate(T* buffer);

template <class T1, class T2>
inline void construct(T1* p, const T2& value) {
       new (p) T1(value);
}

template <class T>
inline void destroy(T* pointer) {
       pointer->~T();
}
```

`deallocate` frees the buffer allocated by `allocate`. For every memory model there are corresponding `deallocate`, `construct` and `destroy` template functions defined with the first argument type being the pointer type of the memory model.

```
template <class T>
pair<T*, ptrdiff_t> get_temporary_buffer(ptrdiff_t n, T*);

template <class T>
void return_temporary_buffer(T* p);
```

`get_temporary_buffer` finds the largest buffer not greater than `n*sizeof(T)`, and returns a pair consisting of the address and the capacity (in the units of `sizeof(T)`) of the buffer. `return_temporary_buffer` returns the buffer allocated by `get_temporary_buffer`.

13 Bibliography

M. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, Massachusetts, 1990.

D. Kapur, D. R. Musser, and A. A. Stepanov, “Tecton, A Language for Manipulating Generic Objects,” *Proc. of Workshop on Program Specification*, Aarhus, Denmark, August 1981, *Lecture Notes in Computer Science*, Springer-Verlag, vol. 134, 1982.

D. Kapur, D. R. Musser, and A. A. Stepanov, “Operators and Algebraic Structures,” *Proc. of the Conference on Functional Programming Languages and Computer Architecture*, Portsmouth, New Hampshire, October 1981.

A. Kershenbaum, D. R. Musser, and A. A. Stepanov, “Higher Order Imperative Programming,” Technical Report 88-10, Rensselaer Polytechnic Institute, April 1988.

A. Koenig, “Associative arrays in C++,” *Proc. USENIX Conference*, San Francisco, CA, June 1988.

A. Koenig, “Applicators, Manipulators, and Function Objects,” *C++ Journal*, vol. 1, #1, Summer 1990.

D. R. Musser and A. A. Stepanov, “A Library of Generic Algorithms in Ada,” *Proc. of 1987 ACM SIGAda International Conference*, Boston, December, 1987.

D. R. Musser and A. A. Stepanov, “Generic Programming,” invited paper, in P. Gianni, Ed., *ISSAC ‘88 Symbolic and Algebraic Computation Proceedings, Lecture Notes in Computer Science*, Springer-Verlag, vol. 358, 1989.

D. R. Musser and A. A. Stepanov, *Ada Generic Library*, Springer-Verlag, 1989.

D. R. Musser and A. A. Stepanov, “Algorithm-Oriented Generic Libraries,” *Software Practice and Experience*, vol. 24(7), July 1994.

M. Stahl and U. Steinmüller, “Generic Dynamic Arrays,” *The C++ Report*, October 1993.

J. E. Shopiro, “Strings and Lists for C++,” *AT&T Bell Labs Internal Technical Memorandum*, July 1985.

A. A. Stepanov and M. Lee, “The Standard Template Library,” Technical Report HPL-94-34, Hewlett-Packard Laboratories, April 1994.

B. Stroustrup, *The Design and Evolution of* C++, Addison-Wesley, Massachusetts, 1994.