# A Short Introduction to the STL

Geoff Kuenning

December 4, 2002

## 1 Introduction

STL, the "Standard Template Library," is a library of C++ templates developed at HP Laboratories by Alex Stepanov and others. For a number of reasons, STL has been gaining popularity, and in the past few years it received a major boost when the ANSI C++ committee decided to make it part of the C++ standard. Because of the standardization, you can expect that STL will continue to increase in importance in the future.

The material in this document is taken primarily from three sources:

1. *STL Tutorial and Reference Guide*, by David R. Musser and Atul Saini, Addison-Wesley, 1996.
2. *The C++ Programming Language, Third Edition* by Bjarne Stroustrup, Addison-Wesley, 1997.
3. The draft C++ standard, available from:
   `ftp://research.att.com/dist/c++std/WP/`.

There is also a wealth of information available from the STL resource page at:

`http://www.cyberport.com/~tangent/programming/stl/resources.html`

## 2 A Quick Overview of STL

The essence of STL can be found in its commitment to *generality* and *flexibility*. As a rule, you can use any STL data structure to store any type (including built-in types and other STL data structures), and you can use any of STL's built-in algorithms to process any data structure.

For example, you can create an array of integers in the normal fashion, and then use STL's sort algorithm to sort them:

```
int mystuff[100];
// fill mystuff...
sort(&mystuff[0], &mystuff[100]);
```

This is simple to write, and simple to understand. The strength of STL is that you can use the same approach on other things. For example:

```
class Simple {
    // standard stuff
    bool operator<(const Simple& right);
    // private data
};
Simple mystuff[100];
// fill mystuff...
sort(&mystuff[0], &mystuff[100]);
```

```
/*
 * Do it again with self-expanding vectors.  The constructor
 * syntax will copy mystuff into morestuff.  This time, we
 * will sort into reverse order.
 */
vector<Simple> morestuff(&mystuff[0], &mystuff[100]);
sort(morestuff.begin(), morestuff.end(), greater<Simple>());
```

If you decided you wanted to use something other than a self-expanding vector, you could simply replace `vector` with the name of some other data structure such as `deque`. STL has a number of built-in algorithms besides `sort`: `find`, `merge`, and `reverse` are only a few of them.

STL also provides generalized iterators. The constructs `morestuff.begin()` and `morestuff.end()` above are actually special cases of iterators. More generally, any data structure has a built-in iterator for walking it. For example:

```
vector<double> doublestuff;
// Stick some things in doublestuff...
vector<double>::iterator i = doublestuff.begin();
while (i != doublestuff.end()) {
    cout << *i << ' ';
    i++;
}
cout << endl;
```

Here, `i` is an iterator that will walk through the vector `doublestuff`. Iterators can also be copied and passed as parameters, and other operations may be possible depending on the particular data structure.

Actually, the above example can be done more concisely, because there are iterators that operate on I/O streams:

```
vector<double> doublestuff;
// Stick some things in doublestuff...
copy(doublestuff.begin(), doublestuff.end(),
  ostream_iterator(cout, ' '));
cout << endl;
```

Search algorithms such as `find` return iterators, so you can do things like this:

```
vector<double>::iterator i;
i = find(doublestuff.begin(), doublestuff.end(), 1.2);
doublestuff.erase(i, i + 3);
```

This erases (deletes) the first occurrence of the value 1.2, and the two following values as well. The remaining vector elements will shift down to fill the hole.

STL provides a number of data structures, all of which are collectively referred to as *containers*. The complete list includes vectors (infinite arrays), deques, lists, sets (balanced binary trees that store only keys), multisets (sets that can handle multiple occurrences of the same key), maps (balanced binary trees that store key/data pairs), and multimaps (maps that can handle multiple occurrences of the same key).

There are also several *adaptors* that modify containers to behave as other useful structures: stacks, queues, and priority queues.

## 3 Drawbacks of STL

The above summary makes STL look pretty nifty. If you use STL, you will never again have to implement fundamental data structures. A single coding style will allow you to use STL's generic algorithms on any data. STL is designed for (time) efficiency, so you can usually be sure that things will run pretty fast.

So where are the cockroaches hiding? Well, STL is far from perfect. My personal belief is that the ANSI committee was grossly premature in deciding to standardize on this library to the exclusion of all others. While STL has some strong points, it also has some severe weaknesses. In particular:

- Several of the so-called "abstract" data structures are actually overspecified. Deques cannot be implemented as either circularly buffered arrays or linked lists. Sets and maps *must* be implemented as balanced binary trees; I think it would be more honest to call them such.

- Some of the important underlying structure is lost by the abstractions. By calling a binary tree a "set" or "map," STL has created an abstraction that denies the essential tree-ness of the data. This in turn prohibits the library from providing pre-, post-, and level-order iterators. Only an inorder iterator is available.

- The set of built-in algorithms is rather odd. Some are very obviously useful, such as `find` and `sort`. Others seem to have been concocted primarily so that they could be used for examples in documentation: `reverse` reverses the order of a container, which is something that is very rarely useful in practice. Still others are useful, but not often enough to be provided as part of every C++ library in the world, such as partial sum (the sum of every subsequence starting with element zero) and inner product.

- The library does not (yet) provide a hash-table container. Given the utility of this common data structure, this is a glaring and painful omission.

- The library only supports doubly-linked lists. While double lists are usually the right choice, there are situations (e.g., stacks) in which the only features required can be supported by singly-linked lists, and the space occupied by the extra pointer is significant.

- The library causes serious code bloat. Practically everything is templated, often with multiple template arguments. It is very easy to accidentally create dozens of copies of the same code. For example, sorting a vector of integers, a vector of doubles, and a deque of integers will cause three copies of the sort algorithm to be generated, and this is a lot of code. Contrast this with the C approach of using a single `qsort` routine, which is somewhat more difficult to use but can efficiently sort anything it is given.

  Even worse, there are many situations where you will find yourself creating copies of templated routines and classes without even realizing it. For example, in the reverse-sort algorithm given above, `greater<Simple>()` creates a whole templated class. You can even create a class without using angle brackets or calling an overloaded function.

- The STL approach requires frequent comparison of iterators. In the example using `erase` above, it should be clear that the `erase` routine needs to repeatedly increment the iterator passed as the first argument and compare it to the second argument. There are some clever optimizations that can be done in certain cases, but the reality is that STL will spend a lot of time doing these relatively expensive comparisons.

- Many of the algorithms are expensively overgeneralized. For example, every container provides a function that allows "erasure" (removal) of a range of elements from the middle. Removal of even a single element from the middle doesn't always make sense (e.g., in vectors), and removal of a range is unusual enough that it is not wise to provide the feature if it prevents optimizing the most common cases: removing only one element, or removing all of them.

- Sometimes the syntax is very confusing. For example, you can count the number of array elements that are not equal to 1 in two lines (three when wrapped to fit this document):

  ```
  int final_count = 0;
  count_if(&a[0], &a[9], bind2nd(not_equal_to<int>(), 1),
    final_count);
  ```

Huh? This is almost completely unreadable. It would be much simpler to just write a `for` loop that compared each array element to 1 and counted the result:

```
int final_count = 0;
for (i = 0;  i < 9;  i++) {
    if (a[i] != 1)
        final_count++
}
```

This takes five lines instead of two, but every C/C++ programmer in the world will understand what's going on.

The last problem with STL is not so much a difficulty with the library itself as with the C++ language. STL is built on very complex templates, and some compilers (e.g., g++) insist on telling you every last ugly detail of the template as part of an error message. For example, consider the following small (wrong) bit of code:

```
list<string> x;
x->push_back("abc");
```

If you stick it inside a program and try to compile it with g++, you'll get the following very unhelpful message (lines have been wrapped to make it fit on the page):

```
stlbad.cc:7: base operand of '->' has non-pointer type 'list<basic_string<char,
string_char_traits<char>,__default_alloc_template<true,0> >,allocator<basic_string<char,
string_char_traits<char>,__default_alloc_template<true,0> > > >'
```

The basic message ("`you used -> instead of .`") is moderately comprehensible, but look at what happened to `list<string>`!

# 4 Elements of STL

STL provides several different sorts of things to help you build programs:

**Containers** are used to hold objects of arbitrary type. All STL containers can hold an arbitrary amount of data and can be accessed in useful ways.

**Container adaptors** modify containers to behave like certain common data types: stacks, queues, and priority queues.

**Algorithms** manipulate containers in useful ways, such as searching and sorting.

**Iterators** are used to access the contents of containers. STL iterators generally behave like pointers.

**Iterator adaptors** modify iterators to run in reverse or do automatic insertion into containers.

**Function objects** allow certain algorithms to be modified to do things like sort in reverse order. You can write your own function objects to extend those provided by STL.

**Function adaptors** modify function objects to allow still more changes in algorithms.

Each of these will be discussed further in following sections.

## 4.1 Containers

An STL container is anything that can hold multiple objects of a given type. STL supports the following containers:

**Array.** The standard C++ array is a full-fledged STL container. The array should be declared using normal C++ syntax, and is subject to all the strengths (e.g., array/pointer equivalence, efficiency) and limitations (e.g., no bounds checking) of C++ arrays.

**Vector.** A `vector` is an array-like object that extends C++ arrays by adding bounds checking and automatic expansion. A vector is declared using the syntax:
`vector`<*type*> *identifier*;
where *type* is a previously declared data type (either built-in or defined by the user) and *identifier* is the name of the vector.

**Deque.** A `deque` is like a vector, except that it is inexpensive to add and remove items at the front as well as the back. A deque is declared using the syntax:
`deque`<*type*> *identifier*;
where *type* is a previously declared data type (either built-in or defined by the user) and *identifier* is the name of the deque.

**List.** A `list` is a standard doubly linked list. It is cheap to add and remove items at any point, but random access is expensive. Because of the latter characteristic, the standard `sort` algorithm does not work on lists. Instead, there is a special `sort` member function. A list is declared using the syntax:
`list`<*type*> *identifier*;
where *type* is a previously declared data type (either built-in or defined by the user) and *identifier* is the name of the list.

**Set.** In the abstract, a `set` is a data structure that can store and retrieve arbitrary keys. Duplicate keys are not allowed. A set supports efficient ($O(logN)$) insertion, lookup, and removal of a key, and traversal of the keys in order. A set is declared using the syntax:
`set`<*key-type*, *compare-type*> *identifier*;
where *key-type* is a previously declared data type (either built-in or defined by the user) and *identifier* is the name of the set. The *compare-type* is used to control key comparison; if omitted it defaults to `less`<*key-type*>, which causes the built-in `operator<` for *key-type* to be used for comparison.

In the concrete, sets are implemented as balanced binary trees (and the specification effectively prohibits any other implementation).

**Multiset.** A `multiset` is identical to a set, except that a particular key may appear more than once. In addition to the functions supported by sets, it is possible to find the range of a duplicate key. A multiset is declared using the syntax:
`multiset`<*key-type*, *compare-type*> *identifier*;
where *key-type* is a previously declared data type (either built-in or defined by the user) and *identifier* is the name of the multiset. The *compare-type* is used to control key comparison; if omitted it defaults to `less`<*key-type*>, which causes the built-in `operator<` for *key-type* to be used for comparison.

**Map.** A `map` is identical to a set, except that each key is associated with a value. In effect, a map is just a set of key/value `pairs`, with the comparison operator defined to compare only the keys. A map is declared using the syntax:
`map`<*key-type*, *value-type*, *compare-type*> *identifier*;
where *key-type* and *value-type* are previously declared data types (either built-in or defined by the user) and *identifier* is the name of the map. The *compare-type* is used to control key comparison; if omitted it defaults to `less`<*key-type*>, which causes the built-in `operator<` for *key-type* to be used for comparison.

**Multimap.** A `multimap` is identical to a multiset, except that each key is associated with a value. In effect, a multimap is just a multiset of key/value `pairs`, with the comparison operator defined to compare only the keys. A multimap is declared using the syntax:

`multimap`<*key-type*, *value-type*, *compare-type*> *identifier*;

where *key-type* and *value-type* are previously declared data types (either built-in or defined by the user) and *identifier* is the name of the multimap. The *compare-type* is used to control key comparison; if omitted it defaults to `less`<*key-type*>, which causes the built-in `operator<` for *key-type* to be used for comparison.

The comparison operators used in sets and maps are classes that only contain an `operator()` member function. For example, suppose our key is of class `Person` and we wish to build a multiset of persons indexed by name. We would define our comparison operator as follows:

```
class PersonCompare {
    public:
        bool operator()(const Person& a, const Person& b)
          const
              {return strcmp(a.name, b.name) < 0;};
};
```

We could then create the multiset using the declaration:

```
multiset<Person, PersonCompare> human_race;
```

If the `Person` class supported an `operator<` that compared only names, we could simplify the above to:

```
multiset<Person> human_race;
```

However, the latter syntax is not yet supported on all compilers.

### 4.1.1 Member Functions

Every container provides some common and some special member functions. The functions common to all containers include null and copy constructors, destructors, simple assignment, all 6 comparison operators, plus the following special functions:

`begin()` Returns an iterator that points to the beginning of the container.

`end()` Returns an iterator that points *just past* the end of the container. This iterator can be used in comparisons to decide whether to terminate a loop.

`rbegin()` Returns a *reverse iterator* that points to the end of the container. When this iterator is incremented using `++`, it will step through the container in reverse order.

`rend()` Returns a reverse iterator that can be used in comparisons to decide whether to terminate a loop that uses a reverse iterator.

`size()` Returns the number of elements in the container.

`max_size()` Returns the `size()` of the largest possible container.

`empty()` Returns `true` if the container is empty.

`swap(b)` Exchanges the contents of two containers (the container it is called on, and `b`) in constant time.

Other member functions are only available in containers that can support them efficiently. These include:

`insert(p, t)` Inserts `t` before the position referenced by the iterator `p`. Supported by all containers. For sorted containers (sets, multisets, maps, and multimaps) the position is only a hint. For maps and multimaps, the type of `t` is `pair<const K, T>` where K and T are the key and value types, respectively.

`insert(t)` Inserts `t` in the container. Supported by sets, multisets, maps, and multimaps.

`erase(p)` Erases (removes) the element referenced by the iterator `p`. Supported by all containers.

`erase(p, q)` Erases (removes) all elements in the range referenced by the iterators `p` and `q`. The element referenced by `q` is not erased. Supported by all containers.

`find(k)` Returns an iterator that references an element with the key `k`, or the `end()` iterator if the key is not found. Supported by sets, multisets, maps, and multimaps. Note that this is more efficient ($O(logn)$) than the generic `find` function.

`count(k)` Returns a count of the number of elements equal to `k`. Supported by sets, multisets, maps, and multimaps.

`lower_bound(k)` Returns an iterator pointing to the first element with a key that is not less than `k`. Supported by sets, multisets, maps, and multimaps.

`upper_bound(k)` Returns an iterator pointing to the first element with a key that is greater than `k`. Supported by sets, multisets, maps, and multimaps.

`equal_range(k)` Returns a `pair` of iterators that are equal to `lower_bound(k)` and `upper_bound(k)`. Supported by sets, multisets, maps, and multimaps.

`capacity()` Returns the largest number of elements that can be stored without invoking reallocation. Supported by vectors.

`reserve(n)` Ensures that the container is large enough to hold at least `n` elements. Supported by vectors.

`operator[](n)` Provides array-like indexed access. Supported by vectors and deques.

`front()` Returns the first element of the container, i.e., the element that the `begin()` iterator refers to. Supported by vectors, deques, and lists.

`back()` Returns the last element of the container. Supported by vectors, deques, and lists.

`push_front(x)` Adds `x` to the front of the container. Supported by deques and lists.

`push_back(x)` Adds `x` to the end of the container. Supported by vectors, deques, and lists.

`pop_back(x)` Removes the last element of the container and returns it. Supported by vectors, deques, and lists.

`pop_front(x)` Removes the last element of the container and returns it. Supported by deques and lists.

`splice(p, c)` Splices the contents of the container `c` before the position referenced by the iterator `p`. Supported by lists.

`remove(x)` Removes all elements equal to `x`. Supported by lists.

`remove_if(pred)` Removes all elements that satisfy the predicate `pred`. Supported by lists.

`merge(c)` Merges the sorted container `c` into the (sorted) container, maintaining the sorted order.

`sort()` Sorts the container. Supported by lists (because the generic `sort` function would be too slow).

Some containers support special constructors (for example, you can initialize a vector with a constant), and there are also some other less-useful member functions not documented here.

### 4.1.2 Container Adaptors

Sometimes, the STL containers are more general than needed for a particular application, and it can be nice to restrict the interface, both for (possible) future efficiency and so that it is clearer what the code does. The following *container adaptors* do this:

**Stack.** A *stack* limits the functionality of a container to allow only push/pop type operations. The member functions include `push`, `pop`, `top`, `empty`, and `size`. A stack is declared using the syntax:

  `stack`<*container*> *identifier*;

where *container* is a container supported by STL and *identifier* is the name of the stack. For example, a `stack` of `ints`, built using a `list` as a base type, would be declared as:

  `stack<list<int> > int_stack;`

**Queue.** A *queue* limits the functionality of a container to allow pushes only at the back and pops only at the front. The member functions include `push`, `pop`, `front`, `back`, `empty`, and `size`. A queue is declared using the syntax:

  `queue`<*container*> *identifier*;

where *container* is a container supported by STL and *identifier* is the name of the queue. For example, a `queue` of `Persons`, built using a `deque` as a base type, would be declared as:

  `queue<deque<Person> > Person_queue;`

**Priority Queue.** A *priority queue* limits the functionality of a container so that the next element available for retrieval is the largest in the container. The member functions include `push`, `pop`, `top`, `empty`, and `size`. A priority_queue is declared using the syntax:

  `priority_queue`<*container*> *identifier*;

where *container* is a container supported by STL and *identifier* is the name of the priority_queue. For example, a `priority_queue` of `ints`, built using a `deque` as a base type, would be declared as:

  `priority_queue<deque<int> > int_priority_queue(less<int>);`

## 4.2 Iterators

Perhaps the most distinctive feature of STL is the *iterator*, which provides unified access to all containers. Iterators are divided into several categories:

**Input iterators.** An input iterator allows items to be read from a container, but does not support modification. As a special case, there is an input iterator that can read directly from an `istream`. Input iterators support construction (including copy construction), assignment, comparison, dereferencing using `*` and `->`, and incrementation using `++` (both pre- and post-incrementation are supported, although there are restrictions on the use of post-incrementation). The dereferencing operators do not support writing.

**Output iterators.** An output iterator allows items to be written to a container, but doesn't support reading. As a special case, there is an output iterator that can write items directly to an `ostream`. Output iterators support construction (including copy construction), assignment, comparison, dereferencing using `*` and `->`, and incrementation using `++` (both pre- and post-incrementation are supported, although there are restrictions on the use of post-incrementation). The dereferencing operators do not support reading.

**Forward iterators.** A forward iterator allows traversal in a single direction. There is no going back once a forward iterator has been incremented. Forward iterators support the same operations as input and output iterators, except that dereferenced types can be both read and written.

**Bidirectional iterators.** A bidirectional iterator extends the capabilities of forward iterators by adding the decrementation (`--`) operator, in both pre- and post-decrement incarnations.

**Random-access iterators.** A random-access iterator operates like an array index. As well as the operations supported by bidirectional iterators, it can be incremented or decremented by any integer value (using either `+` and `-` or `+=` and `-=`). The difference between two random-access iterators is defined and returns an integer of type `Distance`. Random-access iterators can be compared for inequality using `<`, `<=`, `>`, and `>=` (this is not true of any other iterator type). Finally, random-access iterators support subscripting: `a[n]` has the expected result if `a` is a random-access iterator and `n` is an integer.

Most iterators are defined by their classes. For example, the iterator for a `set` is of type `set::iterator`. Thus, we might write:

```
set<Student> student_body;
// ...
for (set<Student>::iterator i = student_body.begin();
  i != student_body.end();
  i++) {
    if (i->year() == Senior)
        graduate(*i);
}
```

In addition, all container classes support a `const_iterator` variant, which is usable with constant containers. As with most `const` things, you should try to use `const_iterator` whenever possible, since it makes your functions usable in more situations.

Stream iterators are not defined in classes. Instead, you declare an input-stream iterator using:

```
istream_iterator i(s);
```

where `s` is a stream (e.g., `cin`). The end-of-stream iterator value is created by using the null constructor.

An output-stream iterator can be constructed in the same manner. There is also a constructor that adds delimiters:

```
ostream_iterator i(s, delim);
```

will cause the single-character delimiter `delim` to be written to the stream `s` after every value is written.

Clever use of stream iterators can allow an entire container to be read or written using a single function call.

### 4.2.1 Iterator adaptors

Iterators can also be modified by applying adaptors. If we want an iterator to run in the opposite direction (e.g., to cause a sort to be descending instead of ascending, we can use the `reverse_iterator` adaptor. The syntax to use a reverse iterator is fairly ugly; see the manual for details. Fortunately, it is rare to explicitly construct a reverse iterator, since the `rbegin()` and `rend()` member functions do this for you.

Finally, there are iterator adaptors that cause insertions as a side effect of using `operator*` (but only when the result of `operator*` is being assigned to). The `back_insert_iterator<Container>` uses the `Container` class's `push_back` member function to do the insertions; there is also a `front_insert_iterator` and an `insert_iterator` that use `push_front` and `insert` member functions in a similar fashion. These are normally used with an "er" notation. For example, we might write:

```
vector<int> vector1;
deque<int> deque1;
// Fill deque1...
copy(deque1.begin(), deque1.end(), vector1.begin());
```

but the above wouldn't work because `vector1` is empty, and writing to an element does not automatically create it. Instead, we must use an insert adaptor:

```
copy(deque1.begin(), deque1.end(), back_inserter(vector1));
```

so that `vector1.push_back()` is called to create new elements that are copies of the deque.

## 4.3 Algorithms

STL provides a number of generic algorithms intended to simplify the task of implementing common operations. Many of these algorithms operate on ranges referenced by iterators; in all cases, the range extends from the element referenced by `first` to the element immediately preceding `last`. The container being operated upon is implicitly specified by the iterators, and both iterators must refer to the same container.

Some of the more useful generic algorithms include:

`for_each(first, last, f)` Applies `f` to each element in the range. itself).

`find(first, last, value)` Finds the first occurrence of `value` in the range, and returns an iterator referencing it or `last` if `value` is not found.

`find_if(first, last, pred)` Finds the first element in the range that satisfies the functional predicate `pred`, and returns an iterator referencing it or `last` if no such element is found.

`count(first, last, value, n)` Counts the number of elements in the range that are equal to `value`, and returns `n` plus the count. There is also a `count_if` function that accepts a functional predicate instead of `value`.

`equal(first1, last1, first2)` Compares elements in the range `first1` to `last1` with those starting at `first2`, and returns `true` if they are equal. A variant accepts a functional predicate if an equality test is not desired.

`search(first1, last1, first2, last2)` Tries to find the subsequence from `first2` to `last2` in the range `first1` to `last1`, in a manner similar to a string search. Returns an iterator referencing the beginning of the found sequence. A variant accepts a functional predicate if an equality test is not desired.

`copy(first1, last1, first2)` Copies the range referenced by `first1` and `last1` into the range beginning with `first2`

`transform(first, last, result, unary_op)` Transforms the given range by applying the unary function `unary_op` to each elements, storing the results beginning at the point referenced by the iterator `result`. Note that `result` does not have to reference the same container type, and in fact can even reference a container that stores a completely different data type. A variant accepts two ranges and a binary operation to generate the result.

`replace(first, last, old_value, new_value)` Replaces every occurrence of `old_value` with `new_value` within the range. A `replace_if` variant accepts a functional predicate instead of `old_value`.

`remove(first, last, value)` Removes every occurrence of `value` within the range. A `remove_if` variant accepts a functional predicate instead of `value`.

`random_shuffle(first, last)` Randomizes the members of the range. A variant accepts an alternate random-number generator.

`sort(first, last)` Sorts the range into ascending order. A variant accepts an alternate comparison function.

`stable_sort(first, last)` Sort the range into ascending order, maintaining the relative ordering of equal keys. A variant accepts an alternate comparison function.

`binary_search(first, last, value)` Performs a binary search for `value` on a sorted container, returning an iterator that references the found value. This works on any container, but is $O(logN)$ only on arrays, vectors, and deques. For sets, multisets, maps, and multimaps, it is better to use the built-in `find` operation than this function.

`merge(first1, last1, first2, last2, result)` Merges two sorted ranges, placing the result into the range beginning at the point referenced by `result`. A variant allows the comparison (ordering) function to be specified.

`inplace_merge(first, middle, last)` Merges the consecutive sorted ranges from `first` to `middle` and from `middle` to `last` into a single sorted sequence. A variant allows the comparison (ordering) function to be specified.

`includes(first1, last1, first2, last2)` Returns `true` if the range from `first2` to `last2` is contained within `first1` through `last1`. A variant allows a different comparison function (other than equality) to be specified. Works only on sets, multiset, maps, and multimaps.

`set_union(first1, last1, first2, last2, result)` Unions two sets, placing the result beginning at `result`. A variant allows a different comparison function (other than equality) to be specified. Works only on sets, multiset, maps, and multimaps. There are also functions to perform `set_intersection`, `set_difference`, and `set_symmetric_difference`.

`min(a, b)` Performs the obvious operation. There is also a `max` function.

`min_element(first, last)` Returns an iterator referencing the minimum element in the range. There is also a `max_element` function.

There are also arithmetic algorithms (e.g., summing a subsequence), operations useful on heaps, and many others. Consult an STL reference guide for the full list.

## 4.4 Function objects

Function objects are a rather strange invention of STL, serving as a way to easily create an overloaded function operating on a particular data type. In general, you can treat a function object as if it were a data type. For example, if you are calling an algorithm that accepts a binary operation, you can specify `plus<float>()` to generate a binary operation that adds two `float`s and returns the result. The following function objects are available:

**Arithmetic functions.** `plus`, `minus`, `times`, `divides`, and `modulus` perform the obvious binary operations. `negate` is a unary operation.

**Comparison functions.** All six comparison functions are provided: `equal_to`, `not_equal_to`, `greater`, `less`, `greater_equal`, and `less_equal`.

**Logical functions.** `logical_and`, `logical_or`, and `logical_not` perform the Boolean functions. There is no `logical_xor`, though it would be relatively easy to define one.

### 4.4.1 Function adaptors

Function adaptors can be used to change the meaning of a function object. They are treated like function objects, except that the arguments they accept are other function objects, not values. For example, the `not1` function adaptor and `bind2nd` function adaptor can be used to locate the first element in an array that is less than or equal to 200:

```
int where = find_if(&array[0], &array[1000],
                    not1(bind2nd(greater<int>(), 200));
```

This example is a bit specious, since it would be better to use `less_equal`, but it shows how function adaptors can be combined. The following adaptors are available:

`not1(pred)` Negates the unary predicate `pred`.

`not2(pred)` Negates the binary predicate `pred`.

`bind1st(op, x)` Converts the binary function `op` into a unary function by making its first argument into a constant. If the binary function is `foo(a, b)`, `bind1st(foo, x)` is roughly equivalent to the code:

```
template <class T>
T& new_foo(const T& y)
{
    return foo(x, y);
}
```

where `x` behaves like a constant with respect to the implementation of `new_foo`.

`bind2nd(op, x)` Converts the binary function `op` into a unary function by making its second argument into a constant. See `bind1st`.

`ptr_fun(func)` Converts a pointer to a unary or binary function into a function adaptor so that it can be used with STL algorithms.