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

Copyright (c) 1994 Hewlett-Packard Company

Permission to use, copy, modify, distribute and sell this document for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation.

1 Introduction . . . 3
2 Structure of the library . . . 3
3 Requirements . . . 4
4 Core components . . . 5
4.1 Operators . . . 5
4.2 Pair . . . 5
5 Iterators . . . 6
5.1 Input iterators . . . 7
5.2 Output iterators . . . 8
5.3 Forward iterators . . . 8
5.4 Bidirectional iterators . . . 9
5.5 Random access iterators . . . 10
5.6 Iterator tags . . . 10
5.6.1 Examples of using iterator tags . . . 11
5.6.2 Library defined primitives . . . 12
5.7 Iterator operations . . . 14
6 Function objects . . . 14
6.1 Base . . . 14
6.2 Arithmetic operations . . . 15
6.3 Comparisons . . . 15
6.4 Logical operations . . . 16
7 Allocators . . . 16
7.1 Allocator requirements . . . 16
7.2 The default allocator . . . 18
8 Containers . . . 18
8.1 Sequences . . . 21
8.1.1 Vector . . . 22
8.1.2 List . . . 26
8.1.3 Deque . . . 28
8.2 Associative containers . . . 30
8.2.1 Set . . . 33
8.2.2 Multiset . . . 34
8.2.3 Map . . . 36
8.2.4 Multimap . . . 38
9 Stream iterators . . . 39
9.1 Istream Iterator . . . 40
9.2 Ostream iterator . . . 40
10 Algorithms . . . 41
10.1 Non-mutating sequence operations . . . 41
10.1.1 For each . . . 41
10.1.2 Find . . . 41
10.1.3 Adjacent find . . . 41
10.1.4 Count . . . 42
10.1.5 Mismatch . . . 42
10.1.6 Equal . . . 42
10.1.7 Search . . . 42
10.2 Mutating sequence operations . . . 43
10.2.1 Copy . . . 43
10.2.2 Swap . . . 43

10.2.3 Transform . . . 44
10.2.4 Replace . . . 44
10.2.5 Fill . . . 44
10.2.6 Generate . . . 44
10.2.7 Remove . . . 45
10.2.8 Unique . . . 45
10.2.9 Reverse . . . 46
10.2.10 Rotate . . . 46
10.2.11 Random shuffle . . . 46
10.2.12 Partitions . . . 47
10.3 Sorting and related operations . . . 47
10.3.1 Sort . . . 47
10.3.2 Nth element . . . 48
10.3.3 Binary search . . . 48
10.3.4 Merge . . . 49
10.3.5 Set operations on sorted structures . . . 50
10.3.6 Heap operations . . . 51
10.3.7 Minimum and maximum . . . 52
10.3.8 Lexicographical comparison . . . 53
10.3.9 Permutation generators . . . 53
10.4 Generalized numeric operations . . . 54
10.4.1 Accumulate . . . 54
10.4.2 Inner product . . . 54
10.4.3 Partial sum . . . 54
10.4.4 Adjacent difference . . . 54

11 Adaptors . . . 55
11.1 Container adaptors . . . 55
11.1.1 Stack . . . 55
11.1.2 Queue . . . 55
11.1.3 Priority queue . . . 56
11.2 Iterator adaptors . . . 57
11.2.1 Reverse iterators . . . 57
11.2.2 Insert iterators . . . 59
11.3 Function adaptors . . . 60
11.3.1 Negators . . . 61
11.3.2 Binders . . . 61
11.3.3 Adaptors for pointers to functions . . . 62

12 Memory Handling Primitives . . . 63

13 Bibliography . . . 63

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

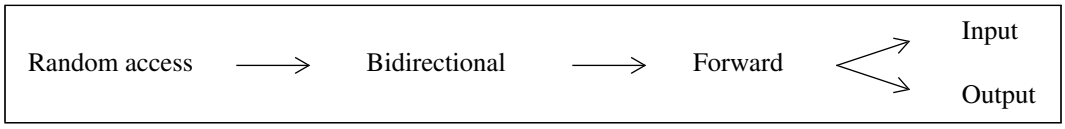
```
return make_pair(5, 3.1415926); // types are deduced

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



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.