C++

Standard library algorithms

Introduction

Container: object holding other objects.

Provides limited set of methods for its manipulation.

To be useful we need more functions to: manipulate its size, iterate, copy, sort and search for elements.

The standard library provides algorithms to perform these basic needs.

Customization of the behavior of these algorithms is done through function objects.

Overview of algorithms in the standard library

- Declarations of the algorithms are found in header <algorithm> in the namespace std.
- Some specific numeric algorithms are declared in header <numeric>.
- Input to algorithms are generally iterators, and eventually function objects.
- Iterators can be: input iterator (In), output iterator (Out), forward iterator (For), bidirectional iterator (Bi), random access iterator (Ran).
- Function object can correspond to: a unary or binary predicate (Pred / BinPred), a unary or binary operation (Op / BinOp), a comparison operator (Cmp).

Overview of algorithms

Algorithms in the standard library can be classified:

- non-modifying sequence operations
- modifying sequence operations
- sorted sequence operations
- set algorithms
- min/max
- heap operations
- permutations

Non-modifying sequence operations

Basic approach to find about elements in a sequence.

Main functions:

- for_each(): apply an operation for each element of a sequence
- find(): find first occurrence of a value in a sequence
- count(): count occurrences of a value in a sequence
- search(): find the first occurrence of a sub-sequence in a sequence

for_each()

for_each() is eliminating loops.

Syntax is: template<class In, class Op> Op for_each(In first, In last, Op f).

It applies f to each element in the sequence [first; last).

for_each()

```
template <class T> class Sum {
 T res;
public:
 void operator() (T x) { res+=x; }
 T result() const { return res; }
};

//...
list<double> l;
l.push_back(1.0); l.push_back(2.0);

Sum<double> s;
s = for_each(l.begin(),l.end(),s);
cout << s.result() << endl; // print 3.0
```

find()

Look through a sequence to find an element matching a value or a predicate.

Syntax:
template <class In, class T> In find(In first, In last, const T& v);
template <class In, class Pred> In find_if(In first, In last, Pred p);

find() returns the iterator *it* to the first element such that **it* == *v*.
If no element is matching, it returns *last.*

find_if() returns the iterator *it* to the first element such that *pred (*it)* is true.
If no element is matching, it returns last.

search()

The search() algorithm finds a sequence as a sub-sequence of another.
Example: "learning" is a sub-sequence of "learning is good".

Syntax:
template <class For, class For2> For search(For first, For last, For2 first2, For2 last2);
template <class For, class For2, class BinPred> For search(For first, For last, For2 first2, For2 last2, BinPred p);

The first version uses ‘==’ to compare the elements, the second uses the binary predicate ‘p’.

search() example

```
#include <string>
#include <iostream>
#include <algorithm>
using namespace std;

int main(void) {
 string quote = "example of a string";
 string::iterator found;
 string sub = "string";
 found = search(quote.begin(), quote.end(),  sub.begin(), sub.
end());
 if (found != quote.end()) cout << "found" << endl;
}
```

Modifying sequence operations

Operations that traverse a sequence and perform a given task.

Some update a sequence in place based on information found during traversal.

Others create a new sequence based on information found during traversal.

Main modifying operations are: copy(), transform(), unique(), replace(), remove(), fill(), generate(), reverse(), rotate() and swap().

Copy()

Produce a new sequence by copying elements from an input sequence.

```
template <class In, class Out> Out copy(In first, In last, Out res);
template <class Bi, class Bi2> Bi2 copy_backward(Bi in, Bi last, Bi2 res);
template <class In, class Out, class Pred> Out copy_if(In first, In last, Out
res, Pred p);
```

copy() copies elements from [first;last) to [res;res+(last-first)).
copy() assumes that elements in [res;res+(last-first)) are dereferenceable.
copy_backward() does not reverse the elements but the order of transfer.
copy_if() adds the possibility to copy an element only if a criterion is met.

Example

The following usage is incorrect because the copy will overwrite the end of the container c.

```
void f(vector<char>& vc) {
 vector<char> c;
 // wrong: overwrite the end of c
 copy(vc.begin(),vc.end(),c.begin());
}
```

Example

The following two approaches work correctly:

```
void f(vector<char>& vc) {
 vector<char> c(vc.size());
 copy(vc.begin(), vc.end(), c.begin()); // ok
}

void f(vector<char>& vc) {
 vector<char> c;
 copy(vc.begin(), vc.end(), back_inserter(c)); // ok
}
```

transform()

Applies an operation to a range of values in a collection and stores the result.

Confusingly, transform() does not change the input but produces a new output.

```
template<class In, class Out, class Op> Out transform(In first, In last, Out res, Op f);
template<class In, class In2, class Out, class BinOp> Out transform(In first, In last, In2 first2, Out res, BinOp  f);
```

The first form applies f to each element of [first;last) and stores the result in [res;res+(last-first)).
The second form applies the binary function f to each element of [first;last) and [first2;first2+(last-first)).

Example

```
template<class Arg, class Res> class Square {
public:
 Res operator()(const Arg& x) { return x*x; }
};

// ...
vector<double> vd;
vd.push_back(1.0); vd.push_back(2.0);
vd.push_back(3.0); vd.push_back(4.0);
typedef vector<double>::value_type value_type;
Square<value_type, value_type> sq;
transform(vd.begin(), vd.end(), vd.begin(), sq);
```

reverse()

Reverses the elements in the container.

Syntax:
template<class Bi> void reverse(Bi first, Bi last);
template<class Bi, class Out> void reverse_copy(Bi first, Bi last, Out res);

reverse() works by swapping the elements within the sequence. It modifies the input sequence.
reverse_copy() does not modify the input sequence and writes the result in [res; res+(last-first)).

Example

```
#include <vector>
#include <iostream>
#include <algorithm>

int main()
{
    int a[] = {1,2,3,4,5,6};
    std::reverse(&a[0], &a[6]);
    std::cout << a[0] << a[1] << a[2]
      << a[3] << a[4] << a[5] << std::endl;
    return 0;
}
```

Result will be: 654321

Sorted sequence operations

It includes algorithms for sorting containers, as well as algorithms for performing operations on sorted sequences.

Operations on sorted sequences include: merge(), binary_search(), lower_bound(), and upper_bound().

To sort sequence we need to compare elements. This is done by a binary predicate.
The default binary predicate is *less()*, which uses '<' by default.
This can be customized.

sort()

sort() is the algorithm sorting the elements of a sequence.

Syntax:
template<class Ran> void sort(Ran first, Ran last);
template<class Ran, class Cmp> void sort(Ran first, Ran last, Cmp cmp);
template<class Ran> void stable_sort(Ran first, Ran last);
template<class Ran, class Cmp> void stable_sort(Ran first, Ran last, Cmp cmp);

Traversal of the container is done with a random access iterator. We can not use *sort()* with a *list* (it provides a bidirectional iterator only).
list has its own *sort()* method (it is a member function).

sort()

Time complexity is O(n * log(n)) on average but can be O(n*n) in worst case.

If a stable sort is required, then stable_sort() should be used.
A sort is *stable* if it preserves the order of equal elements.

Note: list::sort() has worst case complexity of O(n * log(n)).

It is possible to customize the operation used for comparison by specifying it as an argument.

binary_search()

find() allows to search a value in a sequence. However it is slow on large sequence.
If the sequence is sorted, binary_search() is a better algorithm. It finds a given value with a time complexity of O(log(n)).

Syntax:
template<class For, class T> bool binary_search(For first, For last, const T& v);
template<class For, class T, class Cmp> bool binary_search (For first, For last, const T& v, Cmp cmp);

The first version returns true if v is found in the sequence delimited by [first;last).
The second version uses *Cmp* to compare two elements instead of *less*.

lower_bound() and upper_bound()

If there are duplicate elements with the same value, lower_bound() returns an iterator to the first element and upper_bound() returns an iterator to the last-plus-one element for this value.

Syntax:
template<class For, class T> For lower_bound(For first, For last, const T& v);
template<class For, class T> For upper_bound(For first, For last, const T& v);
template<class For, class T, class Cmp> For lower_bound(For first, For last, const T& v, Cmp cmp);
template<class For, class T, class Cmp> For upper_bound(For first, For last, const T& v, Cmp cmp);

merge()

Takes two sorted sequences and produce a new sorted sequence including elements of each input sequence.
merge() is stable.

Syntax:
template<class In, class In2, class Out> void merge(In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp> void merge(In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);

merge() stores the merged sequence in [res;res+(last-first)+(last2-first2)).
The first form uses *less* for comparison while the second uses *Cmp*.

Set operations

Set operations are: union, intersection, difference and checking if an element is in a set.

The set operations provided by the standard library can apply naturally to the containers: *set* and *multiset*.
They can also be applied to any sorted sequence.

Set operations in the standard library assume that the sequence is sorted. Applying set operations to an unsorted sequence is going to produce results that do not conform to the usual set-theoretical rules.

Set operations

includes() tests if each member of the first sequence is a member of the second sequence.
template<class In, class In2> bool includes(In first, In last, In2 first2, In2 last2);

Union and intersection of two sets are produced by set_union() and set_intersection().
template<class In, class In2, class Out> Out set_union(In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out> Out set_intersection(In first, In last, In2 first2, In2 last2, Out res);

Set operations

set_difference() produces a new sequence storing the element in the first but not in the second set.
template<class In, class In2, class Out> Out set_difference(In first, In last, In2 first2, In2 last2, Out res);

Overloaded versions of these functions accept a comparison operator to be used instead of '<', which is the default.

Example

```
string v1 = "abcd";
string v2 = "cdef";

string v3;
set_union(v1.begin(), v1.end(), v2.begin(), v2.end(),
back_inserter(v3)); // v3 is "abcdef"
string v4;
set_intersection(v1.begin(), v1.end(), v2.begin(), v2.end(),
back_inserter(v4(); // v4 is "cd"
string v5;
set_difference(v1.begin(), v1.end(), v2.begin(), v2.end(),
back_inserter(v5)); // v5 is "ab"
```