Part II
Algorithms

Standard algorithms 5

Summary: *Previous chapters describe the basic effects of algorithms on containers. This chapter is a catalog or reference for algorithms.*

Note: A thorough reading of two or three sections of this chapter to learn the structure and a quick leafing through the rest will be sufficient to give rapid access to a suitable algorithm with sample applications. Only in Part III does the combination of algorithms and containers reveal new aspects.

Without especially mentioning it, all the algorithms presented in this chapter are in namespace `std`. They are completely separated from the special implementation of the containers on which they work. They only know iterators which can be used to access the data structures in containers. The iterators must satisfy only a few criteria (see Chapter 2). For this reason, iterators can be both complex objects and simple pointers. Some algorithms bear the same names as container methods. However, because of the different way in which they are used, no confusion will occur.

The complete separation can, however, also have disadvantages: a very generic `find()` algorithm will have to search a container from beginning to end. The complexity is $O(N)$, where N is the number of elements of the container. If the container structure is known, `find()` can be much faster. For example, the complexity of search in a sorted set container is only $O(\log N)$. Therefore, there are several algorithms which under the same name appear both as a generic algorithm and as a member function of a container. Where the situation allows, the made-to-measure member function is to be preferred.

5.1 Copying algorithms

For reasons of speed, some algorithms exist in two variations: the first works directly on the container, the second copies the container. The second variation is always sensible when a copy process is required, for example to keep the original data, and when, with regard to complexity, the algorithm itself is no more expensive than the copy process. Let us look at the different cases:

1. A copy `B` is to be made of container `A`, removing all elements from the copy which

satisfy a given condition, for example all clients with less than 50,000 dollars of turnover. The following alternatives exist:

- copy `A` to `B` and remove all unwanted elements from `B`, or
- copy all elements from `A` to `B`, but only if they satisfy a given criterion.

Both alternatives are of complexity $O(n)$. It is, however, obvious that the second alternative is faster and therefore a copying variation of the algorithm makes sense.

2. A *sorted* copy `B` is to be generated of container `A`. Here too, two possibilities exist:

   - copy `A` to `B` and sort `B`, or
   - take all elements of `A` and insert them sorted into `B`.

   The second possibility is no better than the first one. The sorting process is at least of complexity $O(N \log N)$, thus definitely greater than copying ($O(N)$). Thus, a variation of a sorting algorithm which at the same time copies is simply superfluous. If a copy is required, the first variation can be chosen without any loss of speed.

In the following sections, the copying variations are mentioned, provided they exist. All algorithms which as well as their proper task also generate a copy of a container bear the suffix `_copy` in their names.

5.2 Algorithms with predicates

'Predicate' means a function object (see Section 1.6.3) which is passed to an algorithm and returns a value of type `bool` when it is applied to a dereferenced iterator. The dereferenced iterator is simply a reference to an object stored in the container.

The function object is to determine whether this object has a given property. Only if this question is answered with `true` is the algorithm applied to this object. A general scheme for this is:

```
template <class InputIterator, class Predicate>
void algorithm(InputIterator first,
               InputIterator last,
               Predicate pred) {
    while (first != last) {
        if(pred(*first)) {              // does predicate apply?
            show_it(*first);            // ... or another function
        }
        ++first;
    }
}
```

The `Predicate` class must not alter an object. An example is given on page 89.

Some algorithms that use predicates have a suffix `_if` in their names, others do not. A feature common to all of them is that they expect a predicate in the parameter list.

5.2.1 Algorithms with binary predicates

A binary predicate requires two arguments. This allows you to formulate a condition for two objects in the container, for example a comparison. The algorithm might contain the following kernel:

```
if(binary_pred(*first, *second)) {      // does the predicate apply?
     do_something_with(*first, *second);
     // ...
```

In this sense, you can also use objects of the classes of Table 1.2 as binary predicates. The second parameter of a binary predicate, however, need not be an iterator:

```
template <class InputIterator,
          class binaryPredicate,
          class T>
void another_algorithm(InputIterator first,
                       InputIterator last,
                       binaryPredicate bpred,
                       T aValue)

 {    while (first != last) {
        if(bpred(*first, aValue)) {
            show_it(*first);
        }
        ++first;
    }
}
```

5.3 Nonmutating sequence operations

The algorithms described in this section work on sequences, but do not alter them. With one exception, all algorithms are of complexity $O(N)$, where N is the number of elements in the sequence. The exception is the `search` algorithm.

5.3.1 for_each

The `for_each` algorithm causes a function to be executed on each element of a container. The definition is so short and simple that it is shown in its entirety:

```
template <class InputIterator, class Function>
Function for_each(InputIterator first,
```

```
                    InputIterator last, Function f) {
    while(first != last)
         f(*first++);
    return f;
}
```

The returned object `f` is mostly ignored. However, the returned object can transport data outside the function body, e.g. the maximum value of the iterated sequence. In the following program, the function is the display of an `int` value which, together with the `for_each` algorithm, writes a vector on the standard output.

The class `Function` in the above definition is a placeholder which could as well be the type of a function object. The `Increment` class for incrementing an `int` value is employed in this way. Of course, although `for_each` itself is a nonmutating algorithm, values of the sequence can be changed by the function or function object. Both possibilities are shown below.

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

void display(int x) {                  // nonmutating function
    cout << x << ' ';
}

class Increment {                      // functor class
    public:
       Increment(int i = 1) : howmuch(i) {}
       void operator()(int& x) {       // mutating operator
         x += howmuch;
       }
    private:
       int howmuch;
};

int main() {
    vector<int> v(5);      // vector of 5 zeros
    // v is not changed:
    for_each(v.begin(), v.end(), display);   // 00000
    cout << endl;

    // with Increment constructor
    // v is changed by the functor, not by for_each:
    for_each(v.begin(), v.end(), Increment(2));
    for_each(v.begin(), v.end(), display);   // 22222
    cout << endl;

    // with Increment object
    Increment anIncrement(7);
```

```
    // v is changed by the functor, not by for_each:
    for_each(v.begin(), v.end(), anIncrement);
    for_each(v.begin(), v.end(), display);   // 99999
}
```

In the example, the return value of `for_each()`, the function object, is not used..

5.3.2 find and find_if

There are two kinds of `find()` algorithm: with and without a compulsory predicate (as `find_if()`). It seeks the position in a container, let us call it `C`, at which a given element can be found. The result is an iterator which either points to the position found or is equal to `C.end()`. The prototypes are:

```
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);
```

The way the `find()` algorithm functions is extensively discussed in Section 1.3.4, with corresponding examples on pages 6 ff. Therefore, we will look at only one example for `find_if()`, that is, a `find()` with the predicate. In a sequence of numbers, the first odd number is sought, with the criterion 'odd' checked by means of a function object.

```
// k5/find_if.cpp
#include<algorithm>
#include<vector>
#include<iostream>

void display(int x) {  std::cout << x << ' ';}

class odd {
    public:
       // odd argument yields true
       bool operator()(int x) { return x % 2;}
};

int main() {
    std::vector<int> v(8);

    for(size_t i = 0; i < v.size(); ++i)
        v[i] = 2*i;                        // all even
    v[5] = 99;                             // an odd number
```

```
    // display
    std::for_each(v.begin(), v.end(), display);
    std::cout << std::endl;

    // search for odd number
    std::vector<int>::const_iterator iter
        = std::find_if(v.begin(), v.end(), odd());

    if(iter != v.end()) {
        std::cout << "The first odd number ("
             << *iter
             << ") was found at position "
             << (iter - v.begin())
             << "." << std::endl;
    }
    else std::cout << "No odd number found." << std::endl;
}
```

Alternatively `bind2nd()` can be used, if the header `<functional>` is included:

```
#include<functional>  // don't forget
  // look for odd number
  std::vector<int>::const_iterator iter
         = std::find_if(v.begin(), v.end(),
                  std::bind2nd(std::modulus<int>(),2));
```

5.3.3 find_end

This algorithm finds a subsequence inside a sequence. Neither this nor the following algorithm (`find_first_of()`) is contained in the original version of the STL (Stepanov and Lee (1995), Musser and Saini (1996)), but both have been added to the C++ standard. The prototypes are:

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_end(ForwardIterator1 first1,
                          ForwardIterator1 last1,
                          ForwardIterator2 first2,
                          ForwardIterator2 last2);

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

The interval `[first1, last1)` is the range to be searched; the interval `[first2, last2)` describes the sequence to be sought. The return value is the

last iterator in the search range that points to the beginning of the subsequence. If the subsequence is not found, the algorithm returns `last1`. If the returned iterator is named `i`,

```
*(i+n) == *(first2+n)
```

or

```
pred(*(i+n), *(first2+n)) == true
```

according to the prototype, apply for all `n` in the range 0 to (`last2-first2`). The complexity is $O(N_2 * (N_1 - N_2))$, when N_1 and N_2 are the lengths of the search range and the subsequence to be sought. Example:

```
// k5/find_end.cpp: find a subsequence within a sequence
#include<algorithm>
#include<vector>
#include<iostream>
using namespace std;

int main() {
    vector<int> v(8);
    vector<int> subsequence1(3);

    // initialize vector and subsquences
    for(size_t i = 0; i < v.size(); ++i)
        v[i] = 2*i;                        // all even

    subsequence1[0] = 4;
    subsequence1[1] = 6;
    subsequence1[2] = 8;

    cout << "vector  ";
    for(size_t i = 0; i < v.size(); ++i)
       cout << v[i] << " ";
    cout << endl;

    // search for subsequence 1
    cout << "subsequence1  (";
    for(size_t i = 0; i < subsequence1.size(); ++i)
       cout << subsequence1[i] << " ";
    cout << ")" << endl;
    vector<int>::const_iterator iter
        = find_end(v.begin(), v.end(),
          subsequence1.begin(), subsequence1.end());

    if(iter != v.end()) {
       cout << "is part of the vector. The first occurence"
                " starts at position "
            << (iter - v.begin())
            << "." << endl;
    }
```

```
    else cout << "is not part of the vector." << endl;
}
```

5.3.4 find_first_of

The algorithm finds an element in a subsequence within a sequence. The prototypes are:

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_first_of(ForwardIterator1 first1,
                               ForwardIterator1 last1,
                               ForwardIterator2 first2,
                               ForwardIterator2 last2);

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

The interval `[first1, last1)` is the search range; the interval `[first2, last2)` describes a range of elements to be sought. The return value is the first iterator `i` in the search range which points to an element which is also present in the second range. Assuming that an iterator `j` points to the element in the second range, then

```
*i == *j
```

or

```
pred(*i, *j) == true
```

apply, according to the prototype. If no element of the first range is found in the second range, the algorithm returns `last1`. The complexity is $O(N_1 * N_2)$, when N_1 and N_2 are the range lengths. Example:

```
// excerpt from k5/find_first_of.cpp
      // search for element, which is also in subsequence
      vector<int>::const_iterator iter
          = find_first_of(v.begin(), v.end(),
                  subsequence.begin(), subsequence.end());

      if(iter != v.end()) {
          cout << "Yes. Element " << *iter
               << " is present in both ranges. Its first "
                  "occurrence in the vector is position "
               << (iter - v.begin())
               << "." << endl;
      }
      else cout << "No match." << endl;
```

5.3.5 adjacent_find

Two identical, directly adjacent elements are found with `adjacent_find()`. Here too, two overloaded variations exist – one without and one with binary predicate. The first variation compares the elements by means of the equality operator ==, the second one uses the predicate. The prototypes are:

```
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);
```

The returned iterator points to the first of the two elements, provided that a corresponding pair is found. The first example shows how to find two identical adjacent elements:

```
// k5/adjacent_find.cpp
#include<algorithm>
#include<vector>
#include<iostream>
#include<showseq.h>

int main() {
    std::vector<int> v(8);

    for(size_t i = 0; i < v.size(); ++i)
        v[i] = 2*i;             // even
    v[5] = 99;                  // two identical adjacent elements
    v[6] = 99;
    br_stl::showSequence(v);

    // find identical neighbors
    std::vector<int>::const_iterator iter
        = std::adjacent_find(v.begin(), v.end());

    if(iter != v.end()) {
       std::cout << "The first identical adjacent numbers ("
             << *iter
             << ") were found at position "
             << (iter - v.begin())
             << "." << std::endl;
    }
    else
      std::cout << "No identical adjacent numbers found."
           << std::endl;
}
```

The second example shows the application of a completely different – in the end arbitrary – criterion. A sequence is checked to see whether the second of two adjacent elements is twice as large as the first one:

```
// k5/adjacent_find_1.find.cpp
#include<algorithm>
#include<vector>
#include<iostream>
#include<showseq.h>

class doubled {
    public:
       bool operator()(int a, int b) { return (b == 2*a);}
};

int main() {
    std::vector<int> v(8);

    for(size_t i = 0; i < v.size(); ++i)
        v[i] = i*i;
    v[6] = 2 * v[5];        // twice as large successor
    br_stl::showSequence(v);

    // search for twice as large successor
    std::vector<int>::const_iterator iter
        = std::adjacent_find(v.begin(), v.end(), doubled());

    if(iter != v.end()) {
        std::cout << "The first number ("
             << *iter
             << ") with a twice as large successor"
                " was found at position "
             << (iter - v.begin())
             << "." << std::endl;
    }
    else
      std::cout << "No number with twice as large "
                   "successor found." << std::endl;
}
```

The technique of employing a function object reveals itself as very useful and powerful. In `operator()()`, arbitrarily complex conditions can be formulated without having to change the `main()` program.

5.3.6 count and count_if

These algorithms counts the number of the elements which are equal to a given value `value` or the number of the elements which satisfy a given predicate. The prototypes are

```
template <class InputIterator, class T>
iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last,
              const T& value);

template <class InputIterator, class Predicate>
iterator_traits<InputIterator>::difference_type
count_if(InputIterator first, InputIterator last,
              Predicate pred);
```

The program fragment shows the application, with reference to the vector `v` of the previous examples.

```
std::cout << "There exist "
          << std::count(v.begin(), v.end(), 99)
          << " elements with the value 99." << std::endl;
```

At its construction, the function object of type `myComparison` receives the value with which the comparison is to be made. Here, `count_if()` enters into the action:

```
// #include...  and so on

class myComparison {
    public:
      myComparison(int i): withwhat(i) {}
      bool operator()(int x) { return x == withwhat;}
    private:
      int withwhat;
};

int main() {
    std::vector<int> v(100);
    // initialize v here (omitted)
    std::cout << "There exist "
          << count_if(v.begin(), v.end(), myComparison(99))
          << " elements with the value 99."
          << std::endl;
}
```

An alternative is

```
std::count_if(v.begin(), v.end(),
              std::bind2nd(std::equal_to<int>(),99));
```

5.3.7 mismatch

`mismatch()` checks two containers for matching contents, with one variation using a binary predicate. The prototypes are:

```
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);
```

The algorithm returns a pair of iterators which point to the first position of mismatch in the corresponding containers. If both containers match, the first iterator of the returned pair is equal to `last1`. The following example shows that the containers do not have to be of the same type: here, a `vector` and a `set` are compared. Because of the sorted storage in the `set`, the vector must be sorted as well:

```
// k5/mismatch.cpp
#include<algorithm>
#include<vector>
#include<set>
#include<showseq.h>

int main() {
    std::vector<int> v(8);

    for(size_t i = 0; i < v.size(); ++i)
        v[i] = 2*i;                  // sorted sequence

    std::set<int> s(v.begin(), v.end()); // initialize set with v
    v[3] = 7;                            // insert mismatch
    br_stl::showSequence(v);             // display
    br_stl::showSequence(s);

    // comparison for match with iterator pair 'where'
    std::pair<std::vector<int>::iterator,
              std::set<int>::iterator>
     where = std::mismatch(v.begin(), v.end(), s.begin());

    if(where.first == v.end())
       std::cout << "Match found." << std::endl;
    else
      std::cout << "The first mismatch ("
                << *where.first << " != "
                << *where.second
                << ") was found at position "
```

```
                << (where.first - v.begin())
                << "." << std::endl;
}
```

In the `set`, no index-like position is defined; therefore an expression of the kind `(where.second - s.begin())` is invalid. It is true that `where.second` points to the position of the mismatch in `s`, but the arithmetic is not permitted. If you really need the relative number with reference to the first element in `s`, you can use `distance()`.

The second example checks character sequences, with the simple `mismatch()` finding the first mismatch, whereas `mismatch()` with binary predicate ignores mismatches in upper case and lower case spelling.

```
// k5/mismat_b.cpp
#include<algorithm>
#include<vector>
#include<iostream>
#include<cctype>

class myCharCompare { // tolerates upper/lower case spelling
    public:
      bool operator()(char x, char y) {
          return tolower(x) == tolower(y);
      }
};

int main() {
    char Text1[] = "Algorithms and Data Structures";
    // text with two errors:
    char Text2[] = "Algorithms and data Struktures";

    // copy texts into vector (-1 because of null byte)
    std::vector<char> v1(Text1, Text1 + sizeof(Text1)-1);
    std::vector<char> v2(Text2, Text2 + sizeof(Text2)-1);

    // compare with iterator pair 'where'
        std::pair<std::vector<char>::iterator,
                  std::vector<char>::iterator>
           where = std::mismatch(v1.begin(),
                                 v1.end(), v2.begin());

    if(where.first != v1.end()) {
        std::cout << Text1 << std::endl
                  << Text2 << std::endl;
        std::cout.width(1 + where.first - v1.begin());
        std::cout << "^  first mismatch" << std::endl;
    }

    // compare with predicate
    where = std::mismatch(v1.begin(), v1.end(), v2.begin(),
                  myCharCompare());
```

```
    if(where.first != v1.end()) {
        std::cout << Text1 << std::endl
                  << Text2 << std::endl;
        std::cout.width(1 + where.first - v1.begin());
        std::cout << "^  first mismatch at\n"
                  "tolerance of upper/lower case spelling"
                  << std::endl;
    }
}
```

The specification of output width in connection with the ^ character is used to mark the position visually on screen. A fixed font is assumed.

5.3.8 equal

`equal()` checks two containers for matching contents, with one variation using a binary predicate. Unlike `mismatch()`, however, no position is indicated. As can be seen from the return type `bool`, it checks only whether the containers match or not. The prototypes are:

```
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);
```

When you compare `equal()` with `mismatch()`, you will see a strong similarity: depending on whether `mismatch()` yields a match or not, `equal()` must return the value `true` or `false` (see Exercises 5.1 and 5.2). An application within the program of the previous example might look as follows:

```
if(std::equal(v1.begin(), v1.end(), v2.begin()))
   std::cout  << "equal character strings"   << std::endl;
else
   std::cout  << "unequal character strings" << std::endl;

// remember the negation which saves some writing effort in the program.
if(!std::equal(v1.begin(), v1.end(), v2.begin(),
               myCharCompare()))
      std::cout  << "un";
std::cout << "equal character strings at "
             "tolerance of upper/lower case spelling "
          << std::endl;
```

Exercises

5.1 What would the implementation of `equal()` look like, if it were to use the `mismatch()` algorithm?

5.2 What would the implementation of `equal()` with a binary predicate look like, if it were to use `mismatch()` with a binary predicate?

5.3.9 search

The `search()` algorithm searches a sequence of size N to see whether a second sequence of size G is contained in it. In the worst case, the complexity is $O(NG)$; on average, however, the behavior is better. The return value is an iterator to the position within the first sequence at which the second sequence starts, provided it is contained in the first one. Otherwise, an iterator to the `last1` position of the first sequence is returned. The prototypes are:

```
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);
```

In the example, a sequence of numbers is searched for, inside another sequence of numbers. The binary predicate compares the absolute values of the numbers, ignoring the signs.

```
// k5/search.cpp
#include<algorithm>
#include<vector>
#include<iostream>
#include<cstdlib>
using namespace std;

class AbsIntCompare { // ignore signs
    public:
      bool operator()(int x, int  y) {
          return abs(x) == abs(y);
```

```
    }
};

int main() {
    vector<int> v1(12);
    for(size_t i = 0; i < v1.size(); ++i)
        v1[i] = i;          // 0 1 2 3 4 5 6 7 8 9 10 11

    vector<int> v2(4);
    for(size_t i = 0; i < v2.size(); ++i)
        v2[i] = i + 5;      // 5 6 7 8

    // search for substructure v2 in v1
    vector<int>::const_iterator
        where = search(v1.begin(), v1.end(),
                       v2.begin(), v2.end());

    // if the sequence v2 does not begin with 5, but with a number ≥ 10,
    // the else branch of the if condition is executed.

    if(where != v1.end()) {
        cout << "  v2 is contained in v1 from position "
             << (where - v1.begin())
             << " onward." << endl;
    }
    else
        cout << "  v2 is not contained in v1."
             << endl;

    // put negative numbers into v2
    for(size_t i = 0; i < v2.size(); ++i)
        v2[i] = -(i + 5); // -5 -6 -7 -8

    // search for substructure v2 in v1, ignore signs
    where = search(v1.begin(), v1.end(),
                   v2.begin(), v2.end(),
                   AbsIntCompare());

    if(where != v1.end()) {
        cout << "  v2 is contained in v1 from position "
             << (where - v1.begin())
             << " onward (signs are ignored)."
             << endl;
    }
    else
        cout << "  v2 is not contained in v1."
             << endl;
}
```

Here, with the changed criterion, it is found that `v2` is contained in `v1`.

5.3.10 search_n

The `search_n()` algorithm searches a sequence for a sequence of equal values. The prototypes are:

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

template <class ForwardIterator, class Size, class T,
          class BinaryPredicate>
ForwardIterator search_n(ForwardIterator first,
                         ForwardIterator last,
                         Size count,
                         const T& value,
                         BinaryPredicate binary_pred);
```

The first function returns the iterator to the start of the first sequence with at least `count` values that are equal to `value`. If such a sequence is not found, the function returns `last`. The second function does not check for equality but evaluates the binary predicate. In case of success, `binary_pred(X, value)` must hold for at least `count` consecutive values `X`.

5.4 Mutating sequence operations

If not specified otherwise, the complexity of all algorithms in this section is $O(N)$, where N is the number of moved or altered elements of the sequence.

5.4.1 iota

tip

This algorithm is part of several STL implementations, but not of the C++ standard. It is shown here, because it can from time to time be employed in practice.

Iota is the ninth letter of the Greek alphabet (ι). The corresponding English word 'iota' means 'a very small quantity' or 'an infinitesimal amount.' However, the name has not been chosen for this reason, but is taken from the ι operator of the APL programming language. As an 'Index generator,' the APL instruction $\iota\ n$ supplies a vector with an ascending sequence of the numbers 1 to n. The function itself is rather simple, as can be seen from the definition:

```
// include/iota.h
#ifndef IOTA_H
#define IOTA_H
namespace br_stl {

  template <class ForwardIterator, class T>
  void iota(ForwardIterator first, ForwardIterator last,
```

```
            T value) {
    while(first != last)
         *first++ = value++;
  }
}
#endif
```

All elements in the interval `[first, last)` of a sequence are assigned a value, with the value being increased by one at each iteration. The type `T` for the value can also be a pointer type, so that addresses are incremented. `iota()` is going to be employed in an example in the next section. In the following text it is assumed that iota is contained in an extra header `<iota.h>`.

5.4.2 copy and copy_backward

The `copy()` algorithm copies the elements of a source range into the destination range; copying can start at the beginning or at the end of the ranges (with `copy_backward()`). If the destination range is not to be overwritten, but the copied elements are to be inserted, an insert iterator is chosen as the output iterator, as shown on page 65. Exceptionally, in order to make the functioning clearer, the complete definitions are shown instead of the prototypes:

```
template <class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first,
                    InputIterator last,
                    OutputIterator result) {
    while (first != last) *result++ = *first++;
    return result;
}

template <class BidirectionalIterator1,
          class BidirectionalIterator2>
BidirectionalIterator2 copy_backward(
                    BidirectionalIterator1 first,
                    BidirectionalIterator1 last,
                    BidirectionalIterator2 result) {
    while (first != last) *--result = *--last;
    return result;
}
```

Here too, as usual in the C++ Standard Library, `last` does not denote the position of the last element, but the position after the last element. As Figure 5.1 shows, three cases must be considered:

1. The ranges are completely separated from each other. The ranges can lie in the same or in different containers. `result` points to the beginning of the destination range. `copy()` copies the source range starting with `*first`. The return value is

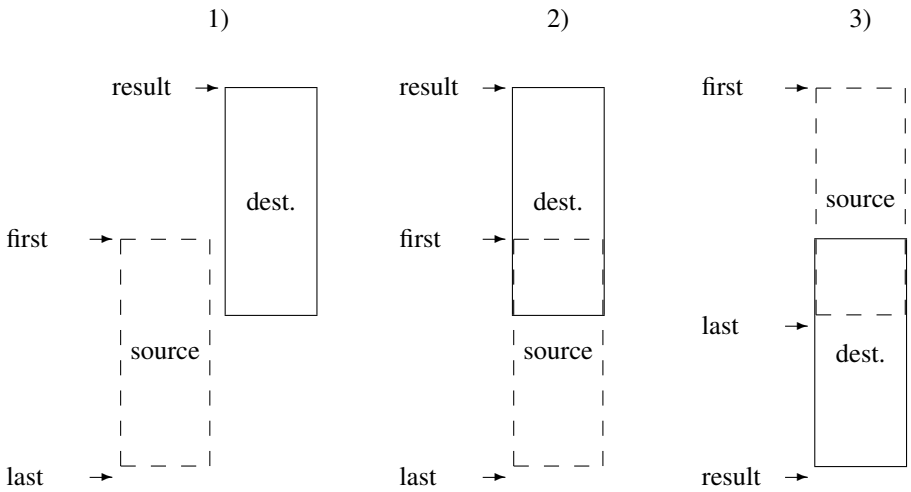

Figure 5.1: Copying without and with range overlapping.

`result + (last - first)`, that is, the position after the last element of the destination range.

2. The ranges overlap in such a way that the destination range begins *before* the source range. `result` points to the beginning of the destination range. `copy()` copies the source range beginning with `*first`. As with the first case, the position after the last element of the destination range is returned.

3. The ranges overlap in such a way that the destination range begins somewhere in the *middle* of the source range. In order not to destroy the data, copying must start from the end. `result` points to the position directly after the *end* of the destination range. `copy_backward()` copies the source range by first copying `*(--last)` to the position `--result`. Here, `result - (last - first)` is returned, that is, the position of the last copied element in the destination range.

The behavior of the copying algorithms is undefined when `result` lies in the interval `[first, last)`. The application of `copy()` and `copy_backward()` is shown in the following example:

```
// k5/cpy.cpp
#include<algorithm>
#include<vector>
#include<iterator>
#include<showseq.h>
#include<iota.h>
using namespace std;
```

```
int main() {
    vector<int> v1(7), v2(7, 0);              // 7 zeros
    br_stl::iota(v1.begin(), v1.end(),0);     // result see next line
    br_stl::showSequence(v1);         // 0 1 2 3 4 5 6
    br_stl::showSequence(v2);         // 0 0 0 0 0 0 0

    /*In the copy process from v1 to v2, the beginning of the destination range is
      marked by v2.begin(). Copy v1 to v2:
    */

    copy(v1.begin(), v1.end(), v2.begin());
    br_stl::showSequence(v2);         // 0 1 2 3 4 5 6

    /*In order to show the variety of the iterator principle, the algorithm copy() is
      used with a special iterator. This iterator is defined as an ostream iterator which
      can display int numbers on the standard output. The copy() algorithm has no
      difficulties with this (in practice, it doesn't give a hoot!).
    */

    // copy v1 to cout, separator *
    ostream_iterator<int> Output(cout, "*");
    copy(v1.begin(), v1.end(), Output);       // 0*1*2*3*4*5*6*
    cout << endl;

    /*Now, a range inside v1 is copied to a different position which lies inside v1. The
      range is chosen such that source and destination ranges overlap. The first four
      numbers are copied, so that case (3) of Figure 5.1 applies.
    */

    // overlapping ranges:
    vector<int>::iterator last = v1.begin();
    advance(last, 4);                         // 4 steps forward
    copy_backward(v1.begin(), last, v1.end());
    copy(v1.begin(), v1.end(), Output);       // 0*1*2*0*1*2*3*
}
```

5.4.3 copy_if

Algorithm `copy_if()` copies all elements of a source range into a destination range, if a certain condition holds.

The algorithm is *not* part of the C++ Standard Library. The reasons are not quite clear because many other algorithms do have a variant with a predicate. Probably the reason is that the same result can be achieved using the algorithm `remove_copy_if()` from page 114, if the predicate is negated. However, it is also very simple to write a standard conforming implementation, as can be seen below. In the example, only values greater than 10 are copied from one container to another.

tip

```
// k5/copy_if.cpp
#include<iostream>
#include<vector>
```

```
#include<functional>
#include<showseq.h>
#include<iota.h>

template <class Iterator1, class Iterator2, class Predicate>
Iterator2 copy_if(Iterator1 iter, Iterator1 sourceEnd,
             Iterator2 destination, Predicate Pred) {

  while(iter != sourceEnd) {
    if(Pred(*iter))
        *destination++ = *iter;
    ++iter;
  }
  return destination;
}

int main() {
  typedef std::vector<int> Container;
  Container V(20);
  br_stl::iota(V.begin(), V.end(), 1);
  br_stl::showSequence(V);

  Container C;         // empty container
  // insert all elements > 10 :
  copy_if(V.begin(), V.end(),
          std::back_inserter(C),
          std::bind2nd(std::greater<int>(),10));
  br_stl::showSequence(C);
}
```

Iterator `destination` must be an insert iterator because the destination container is initially empty. If the destination container has enough room before the algorithm starts, `C.begin()` could be the destination iterator.

5.4.4 swap, iter_swap, and swap_ranges

The `swap()` algorithm exchanges elements of containers or containers themselves. It occurs in three variations:

- `swap()` swaps two individual elements. The two elements can be in the same or in different containers.

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

- `iter_swap()` takes two iterators and swaps the associated elements. The two iterators can belong to the same or to different containers.

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

- `swap_ranges()` swaps two ranges.

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

  `first1` points to the beginning of the first range, `last1` to the position after the last element of the first range. The beginning of the second range is given by `first2`. The number of elements to be swapped is given by the size of the first range. The ranges can lie in the same container, but they must not overlap. `swap_ranges()` returns an iterator to the end of the second range.

- `swap()` is specialized for those container which provide a method `swap()`, i.e. `deque`, `list`, `vector`, `set`, `map`, `multiset`, and `multimap`. These methods are very fast ($O(1)$), because only management information is exchanged. `swap()` calls a container method, as shown here for the swap function specialized for vectors:

```
template<class T, class Allocator>
void swap(vector<T, Allocator>& a,
          vector<T, Allocator>& b) {
   a.swap(b);
}
```

The first three variations are employed in the next example where, for simplicity, all movements take place in the same container – which, in general, is not necessarily the case. At the end of each swapping action, the result is displayed on standard output.

```
// k5/swap.cpp
#include<algorithm>
#include<vector>
#include<showseq.h>
#include<iota.h>
using namespace std;

int main() {
    vector<int> v(17);
    br_stl::iota(v.begin(), v.end(), 10);
    br_stl::showSequence(v);
      // 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26

    cout << "Swap elements v[3] and v[5]:\n";
    swap(v[3], v[5]);                    // swap
    br_stl::showSequence(v);
      // 10 11 12 15 14 13 16 17 18 19 20 21 22 23 24 25 26

    cout << "swap first and last elements"
            " via iterator:\n";
```

```
    vector<int>::iterator first = v.begin(),
                          last  = v.end();
    --last;

    iter_swap(first, last);          // swap
    br_stl::showSequence(v);
      // 26 11 12 15 14 13 16 17 18 19 20 21 22 23 24 25 10

    int oneThird = v.size()/3;
    cout << "swap about the first and last thirds "
         << "(" << oneThird << " Positions):\n";
    last = v.begin();
    advance(last, oneThird);          // end of first third
    vector<int>::iterator target = v.end();
    advance(target, -oneThird);       // beginning of second third

    swap_ranges(first, last, target); // swap
    br_stl::showSequence(v);
      // 22 23 24 25 10 13 16 17 18 19 20 21 26 11 12 15 14
}
```

5.4.5 transform

When the task is not only to copy something, but also to transform it at the same time, then `transform()` is the right algorithm. The transformation can concern only one element or two elements at a time. Correspondingly, there are two overloaded versions:

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

Here, the operation `op` is applied to each element in the range `first` to `last` exclusive, and the result is copied into the range beginning with `result`. `result` may be identical to `first`; in this case, the original elements are substituted by the transformed ones. The return value is an iterator to the position after the end of the target range.

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

In the second version, two ranges are taken into account. The first is the interval `[first1, last1)`, the second the interval `[first2, first2 + last1 - first1)`, that is, the second range has exactly the same size as the first. The `bin_op` operation takes one element from each of the two ranges and stores their result in `result`. `result` may be identical to `first1` or `first2`; in this case, the original elements are substituted with the transformed ones. The return value is an iterator to the position after the end of the target range.

The example shows two vectors of names. The elements of one vector are changed into upper case letters. The elements of the third vector originate from the elements of the first two vectors joined by 'and.'

```
// k5/transform.cpp
#include<algorithm>
#include<showseq.h>
#include<string>
#include<vector>

// unary operation as function
std::string upper_case(std::string s) {
    for(size_t i = 0; i < s.length(); ++i)
        if(s[i] >= 'a' && s[i] <= 'z')
           s[i] -= 'a'-'A';
    return s;
}

class join {                // binary operation as functor
    public:
      std::string operator()(const std::string& a,
                             const std::string& b) {
            return a + " and " + b;
       }
};

int main() {
    vector<string> Gals(3), Guys(3),
                   Couples(3);    // there must be enough space
    Gals[0] = "Annabella";
    Gals[1] = "Scheherazade";
    Gals[2] = "Xaviera";

    Guys[0]  = "Bogey";
    Guys[1]  = "Amadeus";
    Guys[2]  = "Wladimir";

    std::transform(Guys.begin(), Guys.end(),
              Guys.begin(),    // target == source
              upper_case);
```

```
    std::transform(Gals.begin(), Gals.end(),
              Guys.begin(), Couples.begin(),
              join());

    br_stl::showSequence(Couples, "\n");
}
```

Output of the program is:

Annabella and BOGEY
Scheherazade and AMADEUS
Xaviera and WLADIMIR

The example shows different variations:

- The unary transformation `upper_case()` is implemented as a function, the binary one as a functor. This also works the other way round.
- The application of `upper_case()` with the `transform()` algorithm uses the same container to store the results, whereas the binary transformation `join()` stores the results in a different container `Couples`.

5.4.6 replace and variants

The `replace()` algorithm replaces each occurrence of value `old_value` with `new_value` sequentially. Alternatively, a condition-controlled replacement with a unary predicate is possible with `replace_if()`:

```
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);
```

Now, for the first time, we can see the copying variations of algorithms discussed in Section 5.1:

```
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);
```

The copying variations differ in their names by an added _copy. In the following example, all four cases are presented because up to now there has not been one sample program with a copying variation.

```
// k5/replace.cpp
#include<algorithm>
#include<showseq.h>
#include<string>
#include<vector>

// unary predicate as functor
class Citrusfruit {
    public:

       bool operator()(const std::string& a) {
            return  a == "lemon"
                 || a == "orange"
                 || a == "lime";
       }
};

using namespace std;

int main() {
    vector<string> Fruitbasket(3), Crate(3);

    Fruitbasket[0] = "apple";
    Fruitbasket[1] = "orange";
    Fruitbasket[2] = "lemon";
    br_stl::showSequence(Fruitbasket); // apple orange lemon

    cout << "replace: "
            "replace apple with quince:\n";
    replace(Fruitbasket.begin(), Fruitbasket.end(),
            string("apple"), string("quince"));
    br_stl::showSequence(Fruitbasket); // quince orange lemon

    cout << "replace_if: "
            "replace citrus fruits with plums:\n";
    replace_if(Fruitbasket.begin(), Fruitbasket.end(),
```

```
            Citrusfruit(), string("plum"));
    br_stl::showSequence(Fruitbasket); // quince plum plum

    cout << "replace_copy: "
            "copy and replace the plums "
            "with limes:\n";
    replace_copy(Fruitbasket.begin(), Fruitbasket.end(),
            Crate.begin(), string("plum"), string("lime"));
    br_stl::showSequence(Crate);     // quince lime lime

    cout << "replace_copy_if: copy and replace "
            "the citrus fruits with tomatoes:\n";
    replace_copy_if(Crate.begin(), Crate.end(),
            Fruitbasket.begin(), Citrusfruit(),
            string("tomato"));
    br_stl::showSequence(Fruitbasket); // quince tomato tomato
}
```

Since the scheme is always the same, from now on the `_copy` variations of the algorithms will be considered only as prototypes, but not as examples.

5.4.7 fill and fill_n

When a sequence is to be completely or partly initialized with the same values, the `fill()` or `fill_n()` algorithms will help:

```
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);
```

Both are as simple as `iota()` and easy to apply:

```
// k5/fill.cpp
#include<algorithm>
#include<vector>
#include<showseq.h>
using namespace std;

int main() {
    vector<double> v(8);

    // initialize all values with 9.23
    fill(v.begin(), v.end(), 9.23);
    br_stl::showSequence(v);
```

```
    /*fill_n() expects the specification of the number of elements in the sequence
      which are to be initialized with a value and returns an iterator to the end of the
      range. Here, the first half of the sequence is changed, namely initialized with 1.01:
    */
    vector<double>::const_iterator iter =
       fill_n(v.begin(), v.size()/2, 1.01);

    br_stl::showSequence(v);
    cout << "iter is in position = "
         << (iter - v.begin())
         << ", *iter = " << *iter << endl;
}
```

5.4.8 generate and generate_n

A generator in the `generate()` algorithm is a function object or a function which is called without parameters and whose results are assigned one by one to the elements of the sequence. As with `fill()`, there is a variation which expects an iterator pair, and a variation which needs the starting iterator and a number of pieces:

```
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);
```

The example shows both variations, with the generator occurring in two versions as well. The first generator is a function object and generates random numbers, the second one is a function for generating powers of two.

```
// include/myrandom.h
#ifndef MYRANDOM_H
#define MYRANDOM_H
#include<cstdlib>   // rand() and RAND_MAX

namespace br_stl {

class Random {
    public:
      Random(int b): range(b) {}
      // returns an int-random number between 0 and range -1
      int operator()()  {
          return (int)((double)rand()*range/(RAND_MAX+1.0));
      }
    private:
```

```
      int range;
};
}#endif
```

The random function object uses the standard function `rand()` from `<cstdlib>` which generates a value between 0 and `RAND_MAX` which is subsequently normalized to the required range. For further use, the random number generator is packed into an include file and stored in the include directory.

```
// k5/generate.cpp
#include<algorithm>                    // main program
#include<vector>
#include<showseq.h>
#include<myrandom.h>                   // (see above)
using namespace std;

int PowerOfTwo() {   // double value; begin with 1
    static int Value = 1;
    return (Value *= 2)/2;
}

int main() {
    vector<int> v(12);

    br_stl::Random whatAChance(1000);
    generate(v.begin(), v.end(), whatAChance);
    br_stl::showSequence(v);
       // 10 3 335 33 355 217 536 195 700 949 274 444

    generate_n(v.begin(), 10, PowerOfTwo);   // only 10 out of 12!
    br_stl::showSequence(v); // 1 2 4 8 16 32 64 128 256 512 274 444
}
```

5.4.9 remove and variants

The algorithm removes all elements from a sequence which are equal to a value `value` or which satisfy a predicate `pred`. Here, the prototypes are listed including the copying variations:

```
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);
```

```
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);
```

'Removing an element' in practice means that all subsequent elements shift one position to the left. When only one element is removed, the last element is duplicated, because a copy of it is assigned to the preceding position. `remove()` returns an iterator to the now shortened end of the sequence.

It should be noted that the total length of the sequence does not change! No rearrangement of the memory space is carried out. The range between the returned iterator and `end()` only contains meaningless elements.

tip

```
// k5/remove.cpp
#include<iostream>
#include<algorithm>
#include<vector>
#include<iterator>
#include<string>
#include<cstring>
#include<iota.h>

bool isVowel(char c) {
   return std::strchr("aeiouyAEIOUY", c) != 0;
}

using namespace std;

int main() {
    vector<char> v(26);
    // generate alphabet in lower case letters:
    br_stl::iota(v.begin(), v.end(), 'a');
    ostream_iterator<char> Output(cout, "");
    copy(v.begin(), v.end(), Output);
    cout << endl;
    /*Here, the sequence is not displayed by means of showSequence(), because not
      all values between begin() and end() are to be shown, but only the significant
      ones (iterator last).
    */
```

```
    cout << "remove 't': ";
    vector<char>::iterator last =
         remove(v.begin(), v.end(), 't');

    // last = new end after shifting
    // v.end() remains unchanged
    copy(v.begin(), last, Output);
      // abcdefghijklmnopqrsuvwxyz       (t is missing)
    cout << endl;

    last = remove_if(v.begin(), last, isVowel);
    cout << "only consonants left: ";
    copy(v.begin(), last, Output);
      // bcdfghjklmnpqrsvwxz
    cout << endl;

    cout << "complete sequence up to end() with "
            " meaningless rest elements: ";
    copy(v.begin(), v.end(), Output);
      // bcdfghjklmnpqrsvwxzvwxyzz
    cout << endl;
}
```

5.4.10 unique

The `unique()` algorithm deletes identical consecutive elements except one and is already known as the member function of containers (page 55). In addition, it is provided as a function with an additional copying variation:

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

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

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);
```

A simple example shows the first variant. As with `remove()`, shortening the sequence through deletion of the identical adjacent elements does not affect the total length of the sequence. Therefore, here too an iterator to the logical end of the sequence is returned, which is different from the physical end given by `end()`.

```
// k5/unique.cpp
#include<iostream>
#include<algorithm>
#include<vector>
#include<iterator>
using namespace std;

int main() {
    vector<int> v(20);
    // sequence with identical adjacent elements
    for(int i = 0; i < v.size(); ++i)
        v[i] = i/3;

    ostream_iterator<int> Output(cout, " ");
    copy(v.begin(), v.end(), Output);
                            // 0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6
    cout << endl;

    vector<int>::iterator last =
       unique(v.begin(), v.end());
    copy(v.begin(), last, Output); // 0 1 2 3 4 5 6
}
```

The superfluous elements at `last` and behind can be removed with `v.erase(last, v.end())`.

5.4.11 reverse

`reverse()` reverses the order of elements in a sequence: the first shall be last – and vice versa. Since the first element is swapped with the last, the second with the last but one, and so on, a bidirectional iterator is required which can process the sequence starting with the end.

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

template <class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator first,
```

```
                                BidirectionalIterator last,
                                OutputIterator result);
```

The example reverses a character sequence which represents a nonperfect palindrome and a sequence of numbers.

```
// k5/reverse.cpp
#include<algorithm>
#include<showseq.h>
#include<vector>
#include<iota.h>
using namespace std;

int main() {
    char s[] = "Madam";
    vector<char> vc(s, s + sizeof(s)-1); // –1 because of null byte
    br_stl::showSequence(vc);    // Madam

    reverse(vc.begin(), vc.end());
    br_stl::showSequence(vc);    // madaM

    vector<int> vi(10);
    br_stl::iota(vi.begin(), vi.end(), 10);
    br_stl::showSequence(vi);    // 10 11 12 13 14 15 16 17 18 19

    reverse(vi.begin(), vi.end());
    br_stl::showSequence(vi);    // 19 18 17 16 15 14 13 12 11 10
}
```

5.4.12 rotate

This algorithm shifts the elements of a sequence to the left in such a way that those that fall out at the beginning are inserted back at the end.

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

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

The reference document Stepanov and Lee (1995) states in an 'immediately obvious' way, that for each non-negative integer `i < last - first`, an element is moved from position `(first + i)` into position `(first + (i + (last -`

`middle)) % (last - first))`. In other words: `first` and `last` as usual specify the range in which the rotation is to take place. The `middle` iterator points to the element which is to be located at the beginning of the sequence, after the rotation.

The example shows a series of rotations by one element each and a series of rotations by two positions each.

```
// k5/rotate.cpp
#include<showseq.h>
#include<algorithm>
#include<vector>
#include<iota.h>
using namespace std;

int main() {
    vector<int> v(10);
    br_stl::iota(v.begin(), v.end(), 0);

    for(size_t shift = 1; shift < 3; ++shift) {
       cout << "Rotation by " << shift << endl;
       for(int i = 0; i < v.size()/shift; ++i) {
          br_stl::showSequence(v);
          rotate(v.begin(), v.begin() + shift, v.end());
       }
    }
}
```

The program displays:

```
Rotation by 1
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 0
2 3 4 5 6 7 8 9 0 1
...
9 0 1 2 3 4 5 6 7 8
Rotation by 2
0 1 2 3 4 5 6 7 8 9
2 3 4 5 6 7 8 9 0 1
4 5 6 7 8 9 0 1 2 3
...
8 9 0 1 2 3 4 5 6 7
```

Exercise

5.3 Write an algorithm

```
template <class ForwardIterator, class Distance>
void rotate_steps(ForwardIterator first,
```

```
                    ForwardIterator last,
                    Distance steps);
```

making use of `rotate()` which, apart from the iterators for the range, expects the number of rotations `steps`. A negative value of `steps` will rotate the sequence by `steps` positions to the left, a positive value to the right. The value of `steps` can be greater than the length of the sequence. A possible application could be:

```
vector<int> v(10);
br_stl::iota(v.begin(), v.end(), 0);
br_stl::showSequence(v);

cout << "Rotation by -11 (left)" << endl;
rotate_steps(v.begin(), v.end(), -11);
br_stl::showSequence(v);

cout << "Rotation by +1 (right)" << endl;
rotate_steps(v.begin(), v.end(), 1);
br_stl::showSequence(v);
```

The result would be a sequence shifted by 1 (= 11 modulo 10) to the left, cancelled by the subsequent shift to the right.

5.4.13 random_shuffle

This algorithm is used for random shuffling of the order of elements in a sequence that provides random access iterators, for example `vector` or `deque`. It exists in two variations:

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

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

The shuffling of the order will be uniformly distributed; this obviously depends on the random number generator used. The first variation uses an internal random number function, that is, not one specified in ISO/IEC (1998).

It is expected that the random number generator or the random function will take a positive argument `n` of the distance type of the random access iterator used and return a value between 0 and (`n`−1).

For a change, a second random number generator named `RAND` is specified in the example, which has the advantages of being very simple and independent from

system functions. The disadvantage is its short period. But in many cases, this is irrelevant.

```
// include/rand.h
#ifndef RAND_H
#define RAND_H

namespace br_stl {
 class RAND {
    public:
      RAND() : r(1) {}
      int operator()(int X) {
      // returns an int pseudo random number between 0 and X–1
      // period: 2048
          r = (125 * r) % 8192;
          return int(double(r)/8192.0*X);
      }
    private:
      long int r;
 };
}
#endif
```

This simple random number generator may be used more often by including *rand.h* via `#include`. The two random number generators presented up to now differ not only in their algorithms, but also in their application:

- `RAND` is used when the call needs as function object an argument `X`. A value between 0 and (`X` – 1) is returned. The construction of a `RAND` object does not require parameters.
- `Random` (see page 112) does not need a parameter at all. However, during construction of a `Random` object, a number `X` must be specified which defines the range of possible random numbers (0 to `X` – 1).

Depending on the purpose, one or the other variation may be chosen. More sophisticated random number generators can be found in the literature (for example Knuth (1994)). For the examples in this book, the two variations above are sufficient.

```
// k5/rshuffle.cpp
// Example for random_shuffle()
#include<algorithm>
#include<vector>
#include<showseq.h>
#include<iota.h>
#include<rand.h>
using namespace std;
```

```
int main() {
    vector<int> v(12);
    br_stl::iota(v.begin(), v.end(), 0); // 0 1 2 3 4 5 6 7 8 9 10 11

    br_stl::RAND aRAND;
    random_shuffle(v.begin(), v.end(), aRAND);
    br_stl::showSequence(v);           // 1 5 9 8 3 11 2 0 10 6 7 4

    // use of the system-internal random number generator
    random_shuffle(v.begin(), v.end());
    br_stl::showSequence(v);           // 5 4 6 8 7 2 1 3 10 9 11 0
}
```

5.4.14 partition

A sequence can be split with `partition()` into two ranges such that all elements that satisfy a given criterion `pred` lie before all those that do not. The return value is an iterator which points to the beginning of the second range. All elements lying before this iterator satisfy the predicate. A typical application of such a partition can be found in the well-known quicksort algorithm.

The second variation, `stable_partition()`, guarantees in addition that the relative order of the elements within one range is maintained. From a function point of view, this second variation means that the first variation is normally not needed at all. With limited memory, however, the second variation takes slightly longer to run ($O(N \log N)$ instead of $O(N)$, $N = last - first$), so the STL provides both variations. The prototypes are:

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

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

In the example, a randomly ordered sequence is partitioned into positive and negative numbers. Both simple and stable partitions are shown:

```
// k5/partition.cpp
#include<algorithm>
#include<vector>
#include<functional>
#include<showseq.h>
#include<iota.h>
#include<rand.h>      // see page 120
using namespace std;
```

```
int main() {
    vector<int> v(12);
    br_stl::iota(v.begin(), v.end(), -6);
    br_stl::RAND aRAND;
    random_shuffle(v.begin(), v.end(), aRAND);

    vector<int> unstable = v,
                stable = v;

    partition(unstable.begin(), unstable.end(),
              bind2nd(less<int>(),0));
    stable_partition(stable.begin(), stable.end(),
              bind2nd(less<int>(),0));

    cout << "Partition into negative and positive elements";
    cout << endl << "sequence           :";
    br_stl::showSequence(v);        // -5 -1 3 2 -3 5 -4 -6 4 0 1 -2

    cout << "stable partition   :";
    br_stl::showSequence(stable);   // -5 -1 -3 -4 -6 -2 3 2 5 4 0 1

    cout << "unstable partition :";
    // the negative elements are no longer
    // in their original order
    br_stl::showSequence(unstable); // -5 -1 -2 -6 -3 -4 5 2 4 0 1 3
}
```

5.5 Sorting, merging, and related operations

All algorithms described in this section have two variations. One compares elements with the `<` operator, the other uses a function object which shall be called `comp`. Instead of the function object, a function can be used as well.

The function call with the parameters `A` and `B` or the call `comp(A, B)` of the function object yields `true`, if `A < B` applies with regard to the required ordering relation.

5.5.1 sort

The `sort()` algorithm sorts between the iterators `first` and `last`. It is suitable only for containers with random access iterators, such as `vector` or `deque`. Random access to elements of a list is not possible; therefore, the member function `list::sort()` must be employed for lists of type `list`.

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

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

Sorting is not stable, that is, different elements which have the same sorting key may not have the same position in relation to each other in the sorted sequence that they had in the unsorted sequence. The average cost is $O(N \log N)$ with $N =$ `last - first`. No cost estimate is given for the worst case behavior. If the worst case behavior is relevant, however, it is recommended that you use `stable_sort()`.

By looking into the implementation we can see the basic reason for this: `sort()` uses quicksort, which in the worst case has a complexity of $O(N^2)$, depending on the data and the internal partitioning.

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

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

Even in the worst case, the complexity of `stable_sort()` is $O(N \log N)$, if enough memory is available. Otherwise, the cost is at most $O(N(\log N)^2)$. Internally, a merge sort algorithm is used (more about this on page 131), whose time consumption is on average a constant factor of 1.4 higher than that of quicksort. The time increase of 40% is compensated by the excellent worst case behavior and the stability of `stable_sort()`.

The example shows both variations. The random number generator is taken from the previous example. The use of a function instead of the `<` operator is also shown; the ordering criterion in this case is the *integer* part of a `double` number. This leads to elements with the same key but with different values, which are used to show the non-stability of `sort()`.

```
// k5/sort.cpp
#include<algorithm>
#include<vector>
#include<showseq.h>
#include<rand.h>      // see page 120
using namespace std;

bool integer_less(double x, double y) {
    return long(x) < long(y);
}

int main() {
    vector<double> v(17);
    br_stl::RAND aChance;
```

```
    // initialize vector with random values, with
    // many values having the same integer part:

    for(size_t i = 0; i < v.size(); ++i) {
        v[i] =  aChance(3)  + double(aChance(100)/1000.0);
    }

    random_shuffle(v.begin(), v.end(), aChance);


    vector<double> unstable = v,        // auxiliary vectors
                   stable = v;

    cout << "Sequence             :\n";
    br_stl::showSequence(v);
     // 2.022 0.09 0.069 2.097 0.016 1.032 0.086 0.073 2.065 1.081
     // 1.042 0.045 0.042 1.098 1.077 1.07 0.03

    // sorting with < operator:
    stable_sort(stable.begin(), stable.end());
    cout << "\n no difference, because double number "
            "is used as key\n";
    cout << "stable sorting   :\n";
    br_stl::showSequence(stable);
    // 0.016 0.03 0.042 0.045 0.069 0.073 0.086 0.09 1.032 1.042
    // 1.07 1.077 1.081 1.098 2.022 2.065 2.097

    sort(unstable.begin(), unstable.end());
    cout << "unstable sorting :\n";
    br_stl::showSequence(unstable);
    // 0.016 0.03 0.042 0.045 0.069 0.073 0.086 0.09 1.032 1.042
    // 1.07 1.077 1.081 1.098 2.022 2.065 2.097

    // sorting with function instead of < operator:
    unstable = v;
    stable = v;
    cout << "\n differences, because only the int part "
            "is used as key\n";

    stable_sort(stable.begin(), stable.end(),integer_less);
    cout << "stable sorting (integer key)  :\n";
    br_stl::showSequence(stable);
    // 0.09 0.069 0.016 0.086 0.073 0.045 0.042 0.03 1.032 1.081
    // 1.042 1.098 1.077 1.07 2.022 2.097 2.065

    sort(unstable.begin(), unstable.end(), integer_less);
    cout << "unstable sorting (integer key):\n";
    br_stl::showSequence(unstable);
    // 0.03 0.09 0.069 0.016 0.086 0.073 0.045 0.042 1.07 1.032
    // 1.077 1.081 1.042 1.098 2.065 2.097 2.022
}
```

partial_sort

Partial sorting brings the M smallest elements to the front, the rest remains unsorted. The algorithm, however, does not require the number M, but an iterator `middle` to the corresponding position, so that M = `middle - first` applies. The prototypes are:

```
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);
```

The complexity is approximately $O(N \log M)$. The program excerpt for a vector `v` shows the partial sorting. In the result, all elements in the first half are smaller than those in the second half. Furthermore, in the first half the elements are sorted, in the second half they are not.

```
    br_stl::showSequence(v);
    partial_sort(v.begin(), v.begin()+v.size()/2, v.end());
    cout << "half sorted:\n";
    br_stl::showSequence(v);
```

Both variations exist in a copying version, where the iterators `result_first` and `result_last` refer to the target container. The number of sorted elements results from the smaller of the two differences `result_last - result_first` and `last - first`.

```
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);
```

The returned random access iterator points to the end of the described range, that is, to `result_last` or `result_first + (last - first)`, whichever value is smaller.

Exercise

5.4 Complete the sample program from page 123 with instructions that compare the vectors `stable[]` and `unstable[]` and display all element pairs of `v[]` or `stable[]` for which the stability criterion was violated.

5.5.2 nth_element

The nth largest or nth smallest element of a sequence of random access iterators can be found by means of `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);
```

The iterator `nth` is set to the required position, for example, the beginning of the container. After a call of `nth_element()`, the smallest element has been placed in this position. Thus, the order of elements in the container is *changed*. If before the call `nth` points, for example, to the position `v.begin()` + 6, then, after the call, this position contains the seventh smallest element. After the call of the algorithm, only elements that are smaller than or equal to `(*nth)` and all elements to the right of it stand to the left of `nth`.

tip

The average time of the algorithm is linear ($O(N)$). In the present implementation, the time is $O(N^2)$ in the worst but rare case when a partition mechanism similar to quicksort is used.

```
// k5/nth.cpp      Example for nth_element
#include<algorithm>
#include<deque>
#include<showseq.h>
#include<myrandom.h>
#include<functional>        // greater<>
using namespace std;
```

```
int main() {
    deque<int> d(15);
    generate(d.begin(), d.end(), br_stl::Random(1000));
    br_stl::showSequence(d);
    // 840 394 783 798 911 197 335 768 277 553 477 628 364 513 952

    deque<int>::iterator nth = d.begin();
    nth_element(d.begin(), nth, d.end());

    cout << "smallest element:"
         << (*nth)                      // 197
         << endl;

    /*The standard comparison object greater causes the sequence to be reversed. In
      this case, the greatest element is at the first position:
    */
    // here still is nth == d.begin().
    nth_element(d.begin(), nth, d.end(), greater<int>());

    cout << "greatest element  :"
         << (*nth)                      // 952
         << endl;

    /*With the < operator, the greatest element is at the end:
    */
    nth = d.end();
    --nth;              // now points to the last element
    nth_element(d.begin(), nth, d.end());

    cout << "greatest element  :"
         << (*nth)                      // 952
         << endl;

    // assumption for median value: d.size() is odd
    nth = d.begin() + d.size()/2;
    nth_element(d.begin(), nth, d.end());

    cout << "Median value       :"
         << (*nth)                      // 553
         << endl;
}
```

5.5.3 Binary search

All algorithms in this section are variations of a binary search. The way binary search functions was briefly explained on page 17. When it is possible to access a sorted sequence of n elements randomly with a random access iterator, a binary search is very fast. A maximum of $1 + \log_2 n$ accesses are needed to find the element or to determine that it does not exist.

If random access is not possible, for example in a list where you have to travel

from one element to the other in order to find a given one, access time is of the order $O(n)$.

The STL provides four algorithms used in connection with searching and inserting in sorted sequences, which are very similar to each other:

binary_search

```
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);
```

This is the binary search proper. Here and in the following three algorithms (or six, when you include the `Compare` variations), the forward iterator can be substituted with a random access iterator, provided the container allows it. The function returns `true` if the `value` is found.

Only the `<` operator is used, evaluating, in the first variation, the `(!(*i < value) && !(value < *i))` relation (compare with `operator==()` on page 21). `i` is an iterator in the range `[first, last)`. In the second variation `(!comp(*i, value) && !comp(value, *i))` is evaluated accordingly. An example is shown after the next three algorithms.

lower_bound

This algorithm finds the first position where a value `value` can be inserted without violating the ordering. The returned iterator, let us call it `i`, points to this position, so that insertion with `insert(i, value)` is possible without any further search processes. For all iterators `j` in the range `[first, i)` it holds that `*j < value` or `comp(*j, value) == true`. The prototypes are:

```
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);
```

upper_bound

This algorithm finds the *last* position where a value `value` can be inserted without violating the ordering. The returned iterator `i` points to this position, so that rapid insertion is possible with `insert(i, value)`. The prototypes are:

```
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);
```

equal_range

This algorithm determines the largest subrange within which a value `value` can be inserted at an arbitrary position without violating the ordering. Thus, with regard to ordering, this range contains identical values. The elements `p.first` and `p.second` of the returned iterator pair, here `p`, limit the range. For each iterator `k` which satisfies the condition `p.first` $\leq$ `k` $<$ `p.second`, rapid insertion is possible with `insert(k, value)`. The prototypes are:

```
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);
```

The algorithms described above are now shown with the aid of a sample program. Because of its similarity with `lower_bound()`, `upper_bound()` is not included. You must ensure that the container is sorted, since all algorithms in this section make this assumption.

```
// k5/binarysearch.cpp
// Example for binary_search and related algorithms
#include<algorithm>
#include<list>
#include<string>
#include<showseq.h>
using namespace std;
```

```
int main() {
    list<string> Places;
    Places.push_front("Bremen");
    Places.push_front("Paris");
    Places.push_front("Milan");
    Places.push_front("Hamburg");
    Places.sort();                    // important precondition!
    br_stl::showSequence(Places);

    string City;
    cout << "Search/insert which town? ";
    cin >> City;

    if(binary_search(Places.begin(), Places.end(), City))
       cout << City << " exists\n";
    else
       cout << City << " does not yet exist\n";

    // insertion at the correct position
    cout << City << " is inserted:\n";
    list<string>::iterator i =
        lower_bound(Places.begin(), Places.end(), City);
    Places.insert(i, City);
    br_stl::showSequence(Places);

    // range of identical values
    pair<list<string>::const_iterator,
         list<string>::const_iterator>
     p = equal_range(Places.begin(), Places.end(), City);

    // The two iterators of the pair p limit the range in which City occurs:
    list<string>::difference_type n =
            distance(p.first, p.second);
    cout << City << " is contained " << n
         << " times in the list\n";
}
```

5.5.4 Merging

Merging is a method for combining two sorted sequences into one. Step by step, the first elements of both sequences are compared, and the smaller (or the greater, depending on the ordering criterion) element is placed in the output sequence. The prototypes are:

```
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()` assumes an existing output sequence. When one of the two input sequences is exhausted, the remainder of the other one is copied into the output sequence. A brief example will illustrate this:

```
// k5/merge0.cpp
#include<algorithm>
#include<showseq.h>
#include<vector>
#include<iota.h>
using namespace std;


int main() {
    vector<int> v1(6);                // sequence 1
    br_stl::iota(v1.begin(), v1.end(), 0);    // initialize
    br_stl::showSequence(v1);                 // display

    vector<int> v2(10);               // sequence 2
    br_stl::iota(v2.begin(), v2.end(), 0);    // initialize
    br_stl::showSequence(v2);                 // display

    vector<int> result(v1.size()+v2.size()); // sequence 3

    merge(v1.begin(), v1.end(),       // merge
          v2.begin(), v2.end(),
          result.begin());
    br_stl::showSequence(result);             // display
}
```

The result of the program is

```
0 1 2 3 4 5                 (v1)
0 1 2 3 4 5 6 7 8 9         (v2)
0 0 1 1 2 2 3 3 4 4 5 5 6 7 8 9   (result)
```

Thanks to its structure, merging allows very fast sorting with a complexity of $O(N \log N)$ following the recursive scheme:

1. Split list into two halves.
2. If the halves have more than one element, sort both halves with *this procedure* (recursion).
3. Merge both halves into result list.

Obviously, a nonrecursive variation is possible. Sorting is stable. The disadvantage is the additional storage space for the result. For comparison with the above scheme, the merge sort algorithm is now formulated with the means provided by the STL:

```
// k5/mergesort_vec.cpp      Simple example for mergesort()
#include<algorithm>
#include<showseq.h>
#include<vector>
#include<myrandom.h>

template<class ForwardIterator, class OutputIterator>
void mergesort(ForwardIterator first,
               ForwardIterator last,
               OutputIterator result) {
    typename std::iterator_traits<ForwardIterator>::difference_type
                     n    = std::distance(first, last),
                     Half = n/2;
    ForwardIterator Middle = first;
    std::advance(Middle, Half);

    if(Half > 1)             // sort left half, if needed
        mergesort(first, Middle, result);  // recursion

    if(n - Half > 1) {       // sort right half, if needed
        OutputIterator result2 = result;
        std::advance(result2, Half);
        mergesort(Middle, last, result2);  // recursion
    }

    // merge both halves and copy back the result
    OutputIterator End =
          std::merge(first, Middle, Middle, last, result);
    std::copy(result, End, first);
}

int main() {
    std::vector<int> v(20), buffer(20);
    br_stl::Random whatAChance(1000);

    std::generate(v.begin(), v.end(), whatAChance);
    br_stl::showSequence(v);      // random numbers
```

```
    // sort and display
    mergesort(v.begin(), v.end(), buffer.begin());
    br_stl::showSequence(v);      // sorted sequence
}
```

The last two lines of the function can be combined into one, as can often be found in the implementation of the STL, although this will make it more difficult to read:

```
// Merge both halves and copy back the result
copy(result,
     merge(first, Middle, Middle, last, result), first);
```

The advantage of the algorithm described above over `stable_sort()` is that not just containers working with random access iterators can be sorted. Forward iterators are sufficient, so that `v` in the above program can also be a list. It can be filled with `push_front()`. The only condition is that a list `buffer` exists which has at least as many elements as `v`. Only a few changes are needed in `main()`; `mergesort()` remains unchanged:

```
// Excerpt from k5/mergesort_list.cpp
#include<list>

int main() {      // with list instead of vector
    std::list<int> v;
    for(int i = 0; i < 20; ++i)
        v.push_front(0);              // create space

    br_stl::Random whatAChance(1000);
    std::generate(v.begin(), v.end(), whatAChance);
    br_stl::showSequence(v);          // random numbers

    std::list<int> buffer = v;
    mergesort(v.begin(), v.end(), buffer.begin());
    br_stl::showSequence(v);          // sorted sequence
}
```

The 'merge sort' technique is used in a slightly different form when very large files are to be sorted which do not fit into memory, but mass storage can also be used (see Chapter 10).

Merging in place

When sequences are to be merged in place, a buffer must be used. The function `inplace_merge()` merges sequences in such a way that the result replaces the input sequences. The prototypes are:

```
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);
```

The buffer provided is dependent on the implementation.

```
// k5/merge1.cpp
#include<algorithm>
#include<showseq.h>
#include<vector>

int main() {
    std::vector<int> v(16);                 // even number
    int middle = v.size()/2;
    for(int i = 0; i < middle; ++i) {
        v[i]          = 2*i;                // even
        v[middle + i] = 2*i + 1;            // odd
    }
    br_stl::showSequence(v);
    std::inplace_merge(v.begin(), v.begin() + middle,
                       v.end());
    br_stl::showSequence(v);
}
```

Here, the first half of a vector is filled with even numbers, the second half with odd numbers. After the merge, the same vector contains all numbers without explicitly having to specify a result range:

0 2 4 6 8 10 12 14 1 3 5 7 9 11 13 15 before
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 after

5.6 Set operations on sorted structures

This section describes the basic set operations, such as union, intersection, and so on, on *sorted* structures. In the STL, the `set` class is based on sorted structures (see Section 4.4.1). The complexity of the algorithms is $O(N_1 + N_2)$, where N_1 and N_2 denote the number of elements of the sets involved.

The algorithms presented here, which use output iterators, are suitable for set operations only to a limited extent, as explained in Section 5.6.6.

5.6.1 includes

The function `includes` determines whether each element of a second sorted structure S_2 is contained in the first structure S_1. Thus, it checks whether the second

structure is a subset of the first one. The return value is `true`, if $S_2 \subseteq S_1$ holds, otherwise it is `false`. The prototypes are:

```
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);
```

The following example initializes some `set` objects as sorted structures. You could also take simple vectors, provided they are sorted. Since the example is referred to again in subsequent sections, it contains more than is strictly needed for `includes()`.

```
// Excerpt from k5/set_algorithms.cpp
#include <algorithm>
#include<set>
#include<showseq.h>
using namespace std;

int main () {
    int v1[] = {1, 2, 3, 4};
    int v2[] = {0, 1, 2, 3, 4, 5, 7, 99, 13};
    int v3[] = {-2, 5, 12, 7, 33};

    /*initialize sets with the vector contents default comparison object: less<int>()
      (implicit automatic sorting). sizeof v/sizeof *v1 yields the number of elements in v.
    */
    set<int> s1(v1, v1 + sizeof v1/sizeof *v1);
    set<int> s2(v2, v2 + sizeof v2/sizeof *v2);
    set<int> s3(v3, v3 + sizeof v3/sizeof *v3);
       // s3 see next section

    if(includes(s2.begin(), s2.end(),
                s1.begin(), s1.end())) {
        br_stl::showSequence(s1);            // 1 2 3 4
        cout << " is a subset of ";
        br_stl::showSequence(s2);            // 0 1 2 3 4 5 7 99
    }
```

5.6.2 set_union

The function `set_union` builds a sorted structure which contains all the elements that occur in at least one of two other sorted structures S_1 and S_2. Thus, the union of both structures is formed:

$S = S_1 \cup S_2$

The precondition is that the receiving structure provides enough space, or that it is empty and an insert iterator is used as the output iterator (see Section 5.6.6). The prototypes are:

```
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);
```

At the beginning, the result set `Result` (see below) is empty. In the following example, the output iterator must be an insert iterator. For this purpose, the function `inserter()`, which is described on page 66, is included in the parameter list. It returns an insert iterator. The sole use of `Result.begin()` as the output iterator leads to errors. The reasons for this can be found in Section 5.6.6.

```
set<int> Result;           // empty set (s1, s2, s3 as above)

set_union(s1.begin(), s1.end(),
          s3.begin(), s3.end(),
          inserter(Result, Result.begin()));

br_stl::showSequence(s1);              // 1 2 3 4
cout << "  united with ";
br_stl::showSequence(s3);              // –2 5 7 12 33
cout << " yields ";
br_stl::showSequence(Result);          // –2 1 2 3 4 5 7 12 33
```

5.6.3 set_intersection

The function `set_intersection` builds a sorted structure which contains all the elements that occur in both of two other sorted structures S_1 and S_2. Thus, the intersection of both structures is formed:

$S = S_1 \cap S_2$

The conditions described in Section 5.6.6 apply. The prototypes are:

```
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);
```

In order to delete the old results, `clear()` is called. Otherwise, they would be displayed again.

```
Result.clear();                    // empty the set

set_intersection(s2.begin(), s2.end(),
                 s3.begin(), s3.end(),
           inserter(Result, Result.begin()));

br_stl::showSequence(s2);                    // 0 1 2 3 4 5 7 99
cout << " intersected with ";
br_stl::showSequence(s3);                    // –2 5 7 12 33
cout << " yields ";
br_stl::showSequence(Result);                // 5 7
```

5.6.4 set_difference

The function `set_difference` builds a sorted structure which contains all the elements that occur in the first sorted structure S_1, but not in the second sorted structure S_2. Thus, the difference $S_1 - S_2$ of both structures is formed, which is also written as $S_1 \setminus S_2$. The conditions described in Section 5.6.6 apply. The prototypes are:

```
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);
```

The example follows the above pattern:

```
Result.clear();
set_difference(s2.begin(), s2.end(),
               s1.begin(), s1.end(),
           inserter(Result, Result.begin()));

br_stl::showSequence(s2);                  // 0 1 2 3 4 5 7 99
cout << " minus ";
br_stl::showSequence(s1);                  // 1 2 3 4
cout << " yields ";
br_stl::showSequence(Result);              // 0 5 7 99
```

5.6.5 set_symmetric_difference

The function `set_symmetric_difference` builds a sorted structure which contains all the elements that occur either in the first structure S_1 or in a second sorted structure S_2, but not in both. Thus, the symmetric difference of both structures is formed, which is also called 'exclusive-or.' The symmetric difference can be expressed using the previously introduced operations:

$$S = (S_1 - S_2) \cup (S_2 - S_1)$$

or

$$S = (S_1 \cup S_2) - (S_2 \cap S_1)$$

The conditions described in Section 5.6.6 apply. The prototypes are:

```
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);
```

The last example of this kind shows the symmetric difference:

```
Result.clear();

set_symmetric_difference(s2.begin(), s2.end(),
                         s3.begin(), s3.end(),
          inserter(Result, Result.begin()));

br_stl::showSequence(s2);            // 0 1 2 3 4 5 7 99
cout << "  exclusive or ";
br_stl::showSequence(s3);            // –2 5 7 12 33
cout << "yields ";
br_stl::showSequence(Result);        // –2 0 1 2 3 4 12 33 99
```

5.6.6 Conditions and limitations

It was mentioned on page 134 that the algorithms introduced in this section are only to a certain extent suitable for set operations. The reason is that the output iterator must refer to a container that already has enough space. When there is insufficient space, using an insert iterator does not always make sense. *tip*

Let us consider the following example in which the intersection of two sorted structures `v1` and `v2` is to be found and stored in a result vector `result`. We have three possible cases:

1. `result` provides enough space for the result.
2. `result` lacks space.
3. `result` lacks space at the beginning, but an insert iterator is used.

```
// Case 1: everything OK
#include<algorithm>
#include<vector>
#include<showseq.h>
#include<iterator>    // for case 3 (back_insert_iterator)
using namespace std;

int main () {
   vector<int> v1(4);
   vector<int> v2(5);
   vector<int> result(4,0);
```

```
v1[0] = 2; v1[1] = 4; v1[2] = 9; v1[3] = 13;
v2[0] = 1; v2[1] = 2; v2[2] = 9; v2[3] = 13; v2[4] = 43;

vector<int>::iterator last =
   set_intersection (v1.begin(), v1.end(),
                     v2.begin(), v2.end(),
                     result.begin());
br_stl::showSequence(result);                // 2 9 13 0

cout << "only the interesting range: \n";
vector<int>::iterator temp = result.begin();
while(temp != last)
      cout << *temp++ << ' ';                // 2 9 13
cout << endl;
```

The `last` iterator indicates the position after the last element displayed, so that the output can be limited to the interesting range.

```
// Case 2: result1 is too small:
   vector<int> result1(1,0);
   last = set_intersection (v1.begin(), v1.end(),
                            v2.begin(), v2.end(),
                            result1.begin());
```

tip

Here, the result range is too small, so that the program crashes, or worse, the memory area following the result vector is overwritten. This mistake cannot be picked up by using a vector with index check (see Section 9.1), because only pointers are used. Also, the attempt to generate space by using an insert iterator does not lead to a satisfying result:

```
// Case 3: result2 is too small, but an insert iterator is used
   vector<int> result2(1,0);
   back_insert_iterator<vector<int> > where(result2);
   set_intersection (v1.begin(), v1.end(),
                     v2.begin(), v2.end(),
                     where);
   br_stl::showSequence(result2);            // 0 2 9 13
```

The insert iterator appends the elements at the end without considering whether there is still enough space – it simply does not know any better. Given these three cases, it is evident that set operations on sorted structures make sense only under certain conditions:

- Standard containers from Chapter 3: `vector`, `list`, `deque`
  - The result container provides enough space. The disadvantage is that after the end of the result sequence, there are still old values in the container if the space is more than sufficient.
  - The output iterator `where` must not be identical neither with `v1.begin()` nor with `v2.begin()`.

tip

– The result container is empty. In this case, an insert iterator is to be used as the output iterator.

- Associative containers from Section 4.4: `set`, `map`

  An insert iterator has to be used in any case. The contents of an element must not be changed directly, that is, via a reference to the element. This would be the behavior of a non-inserting output iterator, and the ordering within the container and therefore its integrity would be violated.

Thus, some serious thinking has to be done. If the result container is not empty, but also does not provide sufficient space, there is no elegant solution. The reason for this 'flaw' lies in the requirement that the algorithms must also be able to work on simple C-like arrays without being changed. The best thing would be to concentrate only on the result without caring about available space in containers and iterators to be employed. Chapter 6 introduces set operations without the above restrictions.

5.7 Heap algorithms

The priority queue described in Section 4.3 is based on a binary heap. Before we describe the heap algorithms of the STL, let us define the most important features of a heap:

- The N elements of a heap lie in a continuous array on the positions 0 to $N-1$. It is assumed that random access is possible.
- The kind of arrangement of the elements in the array corresponds to a complete binary tree in which all levels are occupied by elements. The only possible exception is the lowest level in which all elements appear on the left-hand side. Figure 5.2 shows the array representation of a heap H of 14 elements, where the circled numbers represent the array indices (*not* the element values). Thus, the element $H[0]$ is always the root, and each element $H[j], (j > 0)$ has a parent node $H[(j-1)/2]$.

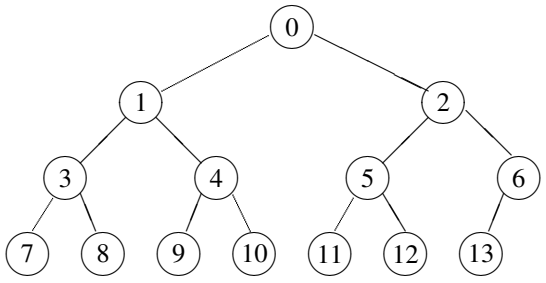

Figure 5.2: Array representation of a heap (number = array index).

- Each element $H[j]$ is assigned a priority which is greater than or equal to the priority of the child nodes $H[2j+1]$ and $H[2j+2]$. For simplicity, we assume that here and in the following discussion large numbers mean high priorities. This could, however, well be the other way round, or completely different criteria might determine the priority. Figure 5.3 shows examples of *element values* of a heap: $H[0]$ equals 99, and so on.

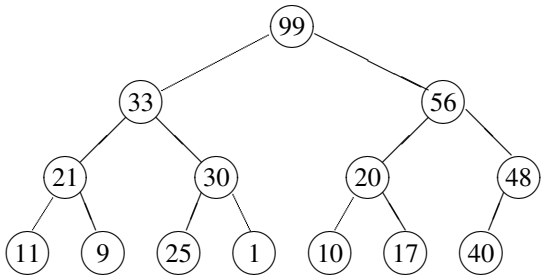

Figure 5.3: Array representation of a heap (number = element value).

Please note that the heap is not completely sorted; we are interested only in the priority relation between parent nodes and corresponding child nodes.

An array H of N elements is a heap if and only if $H[(j-1)/2] \geq H[j]$ holds for $1 \leq j < N$. This means automatically that $H[0]$ is the greatest element. A priority queue simply removes the topmost element of a heap; subsequently, the heap is restructured, that is, the next greatest element moves to the top. With reference to Figures 5.2 and 5.3, this would be element number 2 with the value 56.

The STL provides four heap algorithms which can be applied to all containers that can be accessed with random access iterators:

- `push_heap()` adds an element to an existing heap.
- `pop_heap()` removes the element with the highest priority.
- `make_heap()` arranges all elements in a range in such a way that the range represents a heap.
- `sort_heap()` converts a heap into a sorted sequence.

As usual in the STL, these algorithms do not have to know any details about the containers. They are merely passed two iterators that mark the range to be processed. `less<T>` is predefined as the priority criterion, but a different criterion might be required. Therefore, there is an overloaded variation for each algorithm which allows passing of a comparison object. To show the internal functioning, possible implementations are also shown (compare with the STL of your system).

5.7.1 pop_heap

The function `pop_heap()` removes one element from a heap. The range `[first, last)` is to be considered a valid heap. The prototypes are:

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

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

The 'removal' consists only of the fact that the value with the highest priority, which stands at position `first` is swapped with a value at position (`last` –1). Subsequently, the range `[first, last-1)` is converted into a heap. The complexity of `pop_heap()` is $O(\log(last - first))$.

```
// k5/heap.cpp
#include<algorithm>
#include<showseq.h>
#include<vector>
#include<iota.h>
using namespace std;

int main() {
    vector<int> v(12);                    // container for heap
    br_stl::iota(v.begin(), v.end(), 0); // enter 0 .. 11
    br_stl::showSequence(v);              // 0 1 2 3 4 5 6 7 8 9 10 11

    // create valid heap
    make_heap(v.begin(), v.end()); // see below
    br_stl::showSequence(v);              // 11 10 6 8 9 5 0 7 3 1 4 2

    // display and remove the two numbers
    // with the highest priority:
    vector<int>::iterator last = v.end();
    cout << *v.begin() << endl;    // 11
    pop_heap(v.begin(), last--);

    cout << *v.begin() << endl;    // 10
    pop_heap(v.begin(), last--);
}
```

It should be noted that the end of the heap is no longer indicated by `v.end()`, but by the iterator `last`. With regard to the heap properties of `v` the range between these two values is undefined.

pop_heap implementation

A possible implementation for `pop_heap()` shows how the top element is removed by reorganizing the heap. We assume that the `comp()`-functor behaves like the <-operator, i.e. the biggest number (= high priority) is at the top of the heap. Therefore big numbers are light and small numbers, being at the bottom of the heap, are heavy. If high priorities are represented by *small* numbers, however, the meaning of light and heavy is reversed.

```
// remove top element with pop_heap(first, last--)
template<typename RandomAccessIterator, typename Compare>
void pop_heap(RandomAccessIterator first,
              RandomAccessIterator last,
              Compare comp) {
    iterator_traits<RandomAccessIterator>::
         difference_type size = last - first-1, // new size
                         index = 0,
                         successor = 1;
    assert(size >= 0);

    /*The 'removal' is achieved by first putting the last element at the top position. The
      element that was formerly first is saved at the last position. Saving it is certainly
      not necessary for pop_heap(), but it is cheap and very advantageous if we want
      to sort the heap (see below).
      Then the heap is reorganized by letting the first element sink to its correct place,
      in the course of which first the lighter successors rise and then the sinking element
      is inserted at the position where the successor came from.
    */
    iterator_traits<RandomAccessIterator>::
         value_type temp = *(last - 1); // save last element

    *(last-1) = *first;              // copy first element to the end

    while(successor < size) {
       // perhaps the other successor is more important (i.e. bigger)?
       if(successor+1 < size
          && comp(*(first+successor), *(first+successor+1)))
            ++successor;

       if(comp(temp, *(first+successor))) {
          // follow up
          *(first+index) = *(first+successor);
          index = successor;
          successor = 2*index+1;
       }
       else
          break;
    }
```

```
    // insert element at now free position
    *(first+index) = temp;
}
```

If no compare object is passed, we assume `less<>`:

```
template<typename RandomAccessIterator>
void pop_heap(RandomAccessIterator first,
              RandomAccessIterator last)   {
  pop_heap(first, last, less<
    iterator_traits<RandomAccessIterator>::value_type>());
}
```

5.7.2 push_heap

The function `push_heap()` adds an element to an existing heap. As the prototypes show, the function is passed only two iterators and, if needed, a comparison object. The element to be added does not appear:

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

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

The precondition must apply that the range `[first, last-1)` is a valid heap. `push_heap()` does not care about the value to be added. Therefore, the value to be added to the heap is *previously* entered at its position (`last - 1`). The subsequent call of `push_heap(first, last)` ensures that after the call, the range `[first, last)` is a heap. The handling of this function is somewhat long-winded, but it is only intended as an auxiliary function and it is very fast. The complexity of `push_heap()` is $O(\log(last - first))$. At this point, two numbers are added to the sample heap, as described above:

```
// enter an 'important number' (99)
*last  = 99;
push_heap(v.begin(), ++last);

// enter an 'unimportant number' (–1)
*last  = -1;
push_heap(v.begin(), ++last);

// display of the complete heap
// (no complete ordering, only heap condition!)
br_stl::showSequence(v);    // 99 96 78 50 23 14 –1
```

During insertion, care must be taken that `last` does not run past `v.end()`. Because during removal the value with the highest priority is always placed on top, the output is sorted:

tip

```
// display of all numbers by priority:
while(last != v.begin()) {
   cout << *v.begin() << ' ';
   pop_heap(v.begin(), last--);
}
cout << endl;       // 99 9 8 7 6 5 4 3 2 1 0 –1
```

push_heap implementation

In a `push_heap()` implementation the new element is first inserted at position `last` (see above). From there it goes up to its correct place:

```
// Adding an element value by
// a) placing it at the the last position: *last = value
// b) reorganizing the heap with push_heap(first, ++last)

template<typename RandomAccessIterator, typename Compare>
void push_heap(RandomAccessIterator first,
               RandomAccessIterator last,
               Compare comp) {
    /*The heap is reorganized by letting the last element rise to its correct place, in the
      course of which first the heavier predecessor sinks and then the rising element is
      inserted at the position where the predecessor left. The precondition must apply
      that the range [first, last-1) is a valid heap.
    */

    assert(first < last);
    iterator_traits<RandomAccessIterator>::
         difference_type index = last - first -1,
                         predecessor = (index-1)/2;

    iterator_traits<RandomAccessIterator>::
         value_type temp = *(first+index);       // save element

    while(index != 0     // root not yet reached
          && comp(*(first+predecessor), temp)) {
       // let predecessor sink
       *(first+index) = *(first+predecessor);
       index = predecessor;
       predecessor = (index-1)/2;
    }
    *(first+index) = temp;
}

// without compare object:
template<typename RandomAccessIterator>
```

```
void push_heap(RandomAccessIterator first,
               RandomAccessIterator last)   {
  push_heap(first, last,  less<
    iterator_traits<RandomAccessIterator>::value_type>());
}
```

5.7.3 make_heap

`make_heap()` ensures that the heap condition applies to all elements inside a range. The prototypes are:

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

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

The complexity is proportional to the number of elements between `first` and `last`. The example on page 143 shows the application to a vector as container:

```
make_heap(v.begin(), v.end());     // see page 143
```

make_heap implementation

An implementation for `make_heap()` is easy to write using `push_heap()`, for example:

```
RandomAccessIterator temp = first + 1;
while(temp <= last) push_heap(first, temp++, comp);
```

The complexity is $O(n \log n)$. However, it is possible to make it faster. Only a linear effort is necessary if the heap is constructed bottom up. Beginning with the second level from the bottom, all nodes are compared to their successors. If a successor is greater than the investigated node, their values are exchanged. After this the next level is entered. This process is nothing else than visiting all nodes from the middle to the first in reverse order. That seems to yield a complexity $O(n \log n)$ because after a swap the corresponding subtree has to be checked. But Cormen *et al.* (1994) show that the asymptotical complexity is in fact $O(n)$.

```
// make a heap out of an unsorted array
template<typename RandomAccessIterator, typename Compare>
void make_heap(RandomAccessIterator first,
               RandomAccessIterator last,
               Compare comp) {
   iterator_traits<RandomAccessIterator>::
         difference_type N = last-first,      // size of heap
            i, subroot, left, right, largest;
```

```
    for(i = N/2-1; i >=0 ; --i) {    // begin with the middle element
      largest = i;                   // top of subtree to be checked

      do  // The loop corresponds to the recursive algorithm Heapify() in
      {   // Cormen et al. (1994), Sec. 7.2.
        subroot = largest;           // assumption to be checked
        left  = 2*subroot+1;         // index of left subtree, if it exists
        right = left +1;             // index of right subtree, if it exists

        // compute position of largest element
        if(left<N && comp(*(first+subroot),*(first+left)))
           largest = left;
        else largest = subroot;
        if(right<N && comp(*(first+largest),*(first+right)))
           largest = right;

        if(largest != subroot)       // swap, if heap-condition is violated
           iter_swap(first+subroot, first+largest);
      } while(subroot != largest);   // check heap property for next level
    }
}

// without compare object:
template<typename RandomAccessIterator>
void make_heap(RandomAccessIterator first,
               RandomAccessIterator last)   {
  make_heap(first, last,  less<
    iterator_traits<RandomAccessIterator>::value_type>());
}
```

5.7.4 sort_heap

`sort_heap()` converts a heap into a sorted sequence. The sorting is not stable; the complexity is $O(N \log N)$, when N is the number of elements to be sorted. The prototypes are:

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

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

The sequence is sorted in *ascending* order. This means that the elements of high priority are placed *at the end* of the sequence:

```
// generate new valid heap of all elements
make_heap(v.begin(), v.end());
```

```
    // and sort
    sort_heap(v.begin(), v.end());
    // display of the completely sorted sequence
    br_stl::showSequence(v);    // –1 0 1 2 3 4 5 6 7 8 9 99
```

sort_heap implementation and heapsort

With the help of `pop_heap()`, an implementaion is easy to write:

```
// Sort heap. High priorities (small numbers) lie at
// the beginning when comp = less
template<typename RandomAccessIterator, typename Compare>
void sort_heap(RandomAccessIterator first,
               RandomAccessIterator last,
               Compare comp) {
    // To sort the heap, we successively remove the first element and
    // place it at the current end of the heap (this is done in pop_heap()!)
    while(last - first > 1)
       pop_heap(first, last--, comp);   // remove
}

template<typename RandomAccessIterator>
void sort_heap(RandomAccessIterator first,
               RandomAccessIterator last)   {
  sort_heap(first, last,  less<
    iterator_traits<RandomAccessIterator>::value_type>());
}
```

An arbitrary container with random access (e.g. a vector) can now easily be sorted by converting it into a heap which after this is sorted:

```
// Sort arbitrary vector. Small numbers will be lying at the
// beginning, if comp = less.
// remark: This is not an STL algorithm!
template<typename RandomAccessIterator, typename Compare>
void Heapsort(RandomAccessIterator first,
              RandomAccessIterator last,
              Compare comp) {
   make_heap(first, last, comp);
   sort_heap(first, last, comp);
}

template<typename RandomAccessIterator>
void Heapsort(RandomAccessIterator first,
              RandomAccessIterator last) {
   make_heap(first, last);
   sort_heap(first, last);
}
```

On the average heapsort is slower than quicksort by a factor of about two. However, its complexity $O(n \log n)$ also in the worst case is much better than the worst case complexity of quicksort (average $O(n \log n)$, worst case $O(n^2)$). We will encounter further heap algorithms in Section 11.2.

5.8 Minimum and maximum

The inline templates `min()` and `max()` return the smaller or the greater of two elements, respectively. In case of equality, the first element is returned. The prototypes are:

```
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);
```

The templates `min_element()` and `max_element()` return an iterator to the smallest (or greatest) element of an interval `[first, last)`. In case of equality of the iterators, the first one is returned. The complexity is linear. The prototypes are:

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

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

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

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

5.9 Lexicographical comparison

The lexicographical comparison is used to compare two sequences which can even be of different lengths. The function returns `true` when the first sequence is lexicographically smaller. Both sequences are compared element by element, until the algorithm encounters two different elements. If the element of the first sequence is smaller than the corresponding element of the second sequence, `true` is returned.

If one of the two sequences has been completely searched before a different element is found, the shorter sequence is considered to be smaller. The prototypes are:

```
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);
```

This allows alphabetical sorting of character strings, as shown in the example:

```
// k5/lexicmp.cpp
#include<algorithm>
#include<iostream>
#include<functional>
using namespace std;

char text1[] = "Arthur";
int length1  = sizeof(text1);
char text2[] = "Vera";
int length2  = sizeof(text2);

int main () {
  if(lexicographical_compare(
            text1, text1 + length1,
            text2, text2 + length2))
    cout << text1 << " comes before " << text2 << endl;
  else
    cout << text2 << " comes before " << text1 << endl;

  if(lexicographical_compare(
            text1, text1 + length1,
            text2, text2 + length2,
```

```
            greater<char>()))       // reverse sorting order
    cout << text1 << " comes after " << text2 << endl;
  else
    cout << text2 << " comes after " << text1 << endl;
}
```

The simple `char` arrays are chosen on purpose. We ignore that objects of the `string` class can be compared in this way by means of the < operator. Lexicographical sorting of the kind found in a phone book requires slightly more effort, because, for example, umlauts and accented letters are considered to be equivalent to the corresponding unaccented letters.

5.10 Permutations

A permutation originates from a sequence by exchanging two elements. (0, 2, 1) is a permutation originated from (0, 1, 2). For a sequence of N elements, there exist $N! = N(N-1)(N-2)...2 \cdot 1$ permutations, that is $3 \cdot 2 \cdot 1 = 6$ in the above example:

(0, 1, 2), (0, 2, 1), (1, 0, 2), (1, 2, 0), (2, 0, 1), (2, 1, 0)

You can imagine the set of all $N!$ permutations of a sequence in an ordered form as above, with the ordering created either by means of the < operator or with a comparison object `comp`.

The ordering defines a unique sequence, so that the next or the previous permutation is uniquely determined. The sequence is regarded as cyclic, that is, the permutation following (2, 1, 0) is (0, 1, 2). The algorithms `prev_permutation()` and `next_permutation()` convert a sequence into the previous or next permutation, respectively:

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

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

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

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

When a permutation is found, the return value is `true`. Otherwise, it is the end of a cycle. Then, `false` is returned and the sequence is converted into the smallest possible one (with `next_permutation()`) or the greatest possible one (with `prev_permutation()`), according to the sorting criterion. For example:

```
// k5/permute.cpp
#include<algorithm>
#include<showseq.h>
#include<vector>
#include<iota.h>
using namespace std;

long factorial(unsigned n) {
    long fac = 1;
    while(n > 1) fac *= n--;
    return fac;
}

int main() {
    vector<int> v(4);
    br_stl::iota(v.begin(), v.end(), 0);     //0123
    long fac = factorial(v.size());

    for(int i = 0; i < fac; ++i) {
        if(!prev_permutation(v.begin(), v.end()))
          cout << "Start of cycle:\n";
        br_stl::showSequence(v);
    }
}
```

This example first produces the message 'Start of cycle,' because the initialization of the vector with (0, 1, 2, 3) does not allow determination of a *previous* permutation without exceeding the cycle. Therefore, the greatest sequence after sorting is produced next, namely (3, 2, 1, 0). The 'Start of cycle' message could be prevented by substituting `prev_permutation()` with `next_permutation()` in the example, or alternatively by passing a comparison object `greater<int>()` as the third parameter.

5.11 Numeric algorithms

These algorithms describe general numerical operations. Access to these algorithms is possible via `#include<numeric>`.

5.11.1 accumulate

This algorithm adds all values `*i` of an iterator `i` from `first` to `last` to an initial value. If, instead of the addition, another operation is to be used, there are overloaded variations which are passed this operation as the last parameter. The prototypes are:

```
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 binOp);
```

The following example calculates the sum and the product of all elements of a vector. In these cases, 0 or 1 have to be used as initial values for `init`. Since in the example the vector is initialized with the sequence of natural numbers, the product equals the factorial of 10. The functor `multiplies` is described on page 24.

```
// k5/accumulate.cpp
#include<iota.h>
#include<numeric>
#include<vector>
using namespace std;

int main() {
    vector<int> v(10);
    br_stl::iota(v.begin(), v.end(), 1);

    cout << "Sum = "                              // init + ∑_i v_i
         << accumulate(v.begin(), v.end(), 0)     // 55
         << endl;

    cout << "Product = "
         << accumulate(v.begin(), v.end(), 1L,    // init · ∏_i v_i
                       multiplies<long>())        // 3628800
         << endl;
}
```

5.11.2 inner_product

This algorithm adds the inner product of two containers u and v, which will mostly be vectors, to the initial value `init`:

Result = `init` + $\sum_i v_i \cdot u_i$

Instead of addition and multiplication, other operations may be chosen as well. The prototypes are:

```
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 binOp1,
                binaryOperation2 binOp2);
```

In a Euclidean n-dimensional space R^n, the length of a vector is defined as the root of the inner product of the vector with itself. The example calculates the length of a vector in R^4. The value of `init` must again be 0.

```
// k5/innerproduct.cpp
#include<numeric>
#include<vector>
#include<cmath>
#include<iota.h>
using namespace std;

// functor for calculating the square of a difference (see below)
template<class T>
struct difference_square {
    const T operator()(const T& x, const T& y) {
        const T d = x - y;
        return d*d;
    }
};

int main() {
    int dimension = 4;
    vector<int> v(dimension,1);

    cout << "Length of vector v = "
         << sqrt((double) inner_product(v.begin(), v.end(),
                                        v.begin(), 0))
         << endl;

    /*In order to show the application of other mathematical operators, the following
      part of the example calculates the distance between two points. Besides the func-
      tors of Section 1.6.3, user-defined functors are allowed as well, such as, in this
      case, the functor difference_square.
    */
```

```
    // 2 points p1 and p2
    vector<double> p1(dimension,1.0),
                   p2(dimension);

    br_stl::iota(p2.begin(), p2.end(), 1.0); // arbitrary vector

    cout << "Distance between p1 and p2 = "
         << sqrt( inner_product(p1.begin(), p1.end(),
                                p2.begin(), 0.0,
               plus<double>(),
               difference_square<double>()))
         << endl;
}
```

The first operator is the addition (summation), the second operator the quadrature of the differences:

Distance = $\sqrt{\sum_i (v_i - u_i)^2}$

5.11.3 partial_sum

Partial summation functions in the same way as `accumulate()`, but the result of each step is stored in a result container given by the `result` iterator. The prototypes are:

```
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 binOp);
```

The example shows both variations. The last number of each sequence corresponds to the result of `accumulate()` in the earlier example.

```
// k5/partialsum.cpp
#include<numeric>
#include<vector>
#include<showseq.h>
#include<iota.h>
using namespace std;

int main() {
    vector<long> v(10), ps(10);
    br_stl::iota(v.begin(), v.end(), 1); // natural numbers
```

```
    cout << "vector          = ";
    br_stl::showSequence(v);          // 1 2 3 4 5 6 7 8 9 10

    partial_sum(v.begin(), v.end(), ps.begin());
    cout << "Partial sums    = ";
    br_stl::showSequence(ps);         // 1 3 6 10 15 21 28 36 45 55

    // Sequence of factorials
    cout << "Partial products = ";
    partial_sum(v.begin(), v.end(), v.begin(),
                multiplies<long>());
               // 1 2 6 24 120 720 5040 40320 362880 3628800
    br_stl::showSequence(v);
}
```

5.11.4 adjacent_difference

This algorithm calculates the difference between consecutive elements of a container v and writes the result into a result container e pointed to by the `result` iterator. Since there is exactly one difference value less than there are elements, the first element is retained. If the first element has the index 0, the following holds:

$$e_0 = v_0$$
$$e_i = v_i - v_{i-1}, i > 0$$

Besides calculation of differences, other operations are possible. The prototypes are:

```
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 binOp);
```

The example shows both variations. In the first one, the differences are calculated; in the second one, a sequence of Fibonacci numbers is calculated. (Leonardo of Pisa, called Fibonacci, was an Italian mathematician who lived 1180–1240.)

```
// k5/adjacent_difference.cpp
#include<numeric>
#include<vector>
#include<iota.h>
#include<showseq.h>
using namespace std;
```

```
int main() {
    vector<long> v(10), ad(10);
    br_stl::iota(v.begin(), v.end(), 0);

    cout << "vector          = ";
    br_stl::showSequence(v);            // 0 1 2 3 4 5 6 7 8 9

    cout << "Differences     = ";
    adjacent_difference(v.begin(), v.end(), ad.begin());
    br_stl::showSequence(ad);           // 0 1 1 1 1 1 1 1 1 1

    // Fibonacci numbers
    vector<int> fib(16);
    fib[0] = 1;                         // initial value
    /*One initial value is sufficient here because the first value is written to the first
      position (see formula e_0 = v_0 on the previous page) and the result iterator which
      is shifted by one position (see formula e_i = v_i - v_{i-1} ). Therefore, after the first
      step of the algorithm, fib[1] equals 1.
    */
    cout << "Fibonacci numbers  = ";
    adjacent_difference(fib.begin(), fib.end()-1,
                  (fib.begin()+1), plus<int>());

    br_stl::showSequence(fib);
    // 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
}
```

If, instead of the difference, the sum of both predecessors is used, the result container is filled with a sequence of Fibonacci numbers. Fibonacci asked himself, how many pairs of rabbits there would be after n years, if, beginning with the second year, each couple generates another couple. The fact that rabbits eventually die was ignored for the purpose of this problem. The answer to this question is that the number of rabbits in the year n is equal to the sum of the numbers of the years $n-1$ and $n-2$. Fibonacci numbers play an important role in information science (Knuth (1994), Cormen *et al.* (1994)). It should be noted that at the beginning of the construction of the sequence, the `result` iterator must be `fib.begin()+1`.