Chapter 7

Generic Programming Using the STL

7.1 Generic Programming

Consider the following function template.

```
template <typename T>
const T*
find(const T* first,const T* last,const T& value) {
// sequential search for value in [*first..*last[; return last if not found
while (first!=last && *first!=value)
  ++first;
return first;
}
```

The **find** function implements a sequential search for **value** over the range $[*first \ldots *last[$ of an array. It returns a pointer to where **value** can be found or **last**, if **value** does not appear among $[*first \ldots *last[$. E.g.

```
extern int  a[SIZE];
find(a, a+SIZE, 20);
```

will search the complete array **a**, returning **a+SIZE** if 20 cannot be found.

If the data structure to search is a linked list, sequential search looks similar but not identical.

```
#include <assert.h>
template <typename T>
class Node { // Node* is a (too) simple linked list
public:
  Node(const T& t): value_(t), cdr_(0) {} // you cannot make an empty list
  Node* cons(const T& t) { return new Node(t, this); } // prepend
  const T&  car() const { return value_; } // non const version missing
```

```
  Node* cdr() const { return cdr_; } // non const version missing
private:
  Node(const T& t, Node* cdr): value_(t), cdr_(cdr) {}
  T value_;
  Node* cdr_;
};

template <typename T>
Node<T>*
find(Node<T>* first, Node<T>* last, const T& value) {
// sequential search for value in linked list [first .. last[
while (first!=last && first->car()!=value)
  first = first->cdr();
return first;
}
```

E.g.[1]

```
extern Node<int> *list;
find(list, static_cast<Node<int>*>(0), 20);
```

will search **`list`** for 20, returning **`0`** if it cannot be found.

Both versions of **`find`** implement the same algorithm; the only differences between them come from the idiosyncrasies of the underlying container data structures on which the algorithm operates.

Thus, if we want to implement the sequential search algorithm on 10 different types of container data structures[2], we will end up with 10 slightly different implementations. Worse, if we want to implement n algorithms on m container types, we will need $n \times m$ implementations.

In order to avoid this complexity, we introduce another abstraction that will allow us to have but a single implementation of sequential search which will operate, without modification, on a wide range of container types.

To achieve this, we should consider the essential characteristics of a container type that make it possible to operate a sequential search on it. So let us assume that the container type manages a collection of objects of type **`T`**.

The following is a pseudo-code description of the essence of the sequential search algorithm.

```
Cursor
find(Cursor start,Cursor end, T value) {
while (start!=end && (object-pointed-to-by-start != value) )
  advance start-cursor to next position in container;
return start
}
```

---

[1] Note the use of **`static_cast`** (page 112); it is necessary since template argument deduction attempts very few conversions.

[2] The *standard library* supports more than 10 types of container data structures and more than 70 algorithms.

From this description, we can derive the following requirements[3]:

- The container type must support "cursors" that can be used to refer to a **T**-object in the container. In the array version of **find**, such a cursor is of type **T***, a simple pointer to the object type; in the linked list version, the cursor type is **Node<T>***, i.e. a cursor is a pointer to a node in the linked list.
- The cursor type must support an operation that allows the retrieval of the **T**-object it refers to. Note that we do *not* require that the object a cursor refers to, can be assigned to; **find** only retrieves the object, it does not need to change it. In the array implementation, this operation is the normal pointer dereference **T*::operator*** but in the linked list version, **Node<T>*::operator->().value()** must be called.
- It must be possible to advance a cursor to obtain a new cursor that refers to another element of the container range to be searched. In the array implementation, this is simply achieved using pointer arithmetic (**T*::operator++()**); the linked list version uses **Node<T>*::operator->().cdr()**.
- In order for the algorithm to terminate correctly, it must be possible to compare cursors. Both implementations rely on pointer comparisons for this.
- Finally, cursors must be assignable.

In addition, the type **T** of the objects must support comparison, i.e. **t1!=t2** must have an implementation.

Thus, sequential search can be applied to any container type that has an associated **Cursor** type that supports the above requirements.

If we decide to follow the usual convention for the operation names: ***** for dereference, **++** for advancing and **!=** for testing non-equality, we can summarize the requirements on **Cursor**.

```
class Cursor {
public:
  T  operator*(); // dereference to obtain an object from the container
  Cursor  operator++(); // return cursor that refers to the next object in the container
  bool  operator!=(Cursor); // compare cursors for (non)equality
  Cursor&  operator=(const Cursor&); // cursors must be assignable
};
```

Note that, conveniently, pointer types **T*** satisfy the above requirements[4].

A *generic* implementation of sequential search simply makes the Cursor class a template type argument[5].

[3] Note that most requirements concern the **Cursor** type, not the underlying container type.

[4] Of course, pointers are over-qualified in the sense that they support much more than the minimal requirements.

[5] This is the actual implementation as defined in the standard library.

```
template <class InputIterator, class T>
InputIterator
find(InputIterator first, InputIterator last, const T& value) {
while (first != last && *first != value) ++first;
return first;
}
```

The template argument is called **`InputIterator`** instead of **`Cursor`** because, in the STL jargon, the name **`InputIterator`** stands for the set of requirements mentioned above (see Section 7.2).

The technique where one implements the essence of an algorithm, abstracting from the data types on which it operates by a set of requirements for such data types, is called *generic programming*.

Note that there is no construct in C++ to explicitly represent requirements. Rather we use template type arguments to represent the type that must respect the requirements. As always with template type arguments, the exact requirements can only be seen from the source and an understanding of the algorithm.

Unlike object-oriented programming (Chapter 8), generic programming in C++ does not carry a performance penalty as the price for increased generality and flexibility. Indeed, the use of templates ensures that the binding of a specific type to the requirements occurs at compile-time.

7.2 Iterators

The linked list data structure of the program on page 137 cannot be used as such with the generic sequential search implementation on page 139.

However, it suffices to implement a **`Cursor`** type for the linked list to make it amenable to a generic **`find`**.

```
#include <assert.h>
template <typename T>
class Node { // Node* is a (too) simple linked list
public:
  Node(const T& t): value_(t), cdr_(0) {} // you cannot make an empty list
  Node* cons(const T& t) { return new Node(t, this); } // prepend
  const T& car() const { return value_; } // non const version missing
  Node* cdr() const { return cdr_; } // non const version missing
  class Cursor { // satisfies requirements for find
  public:
    Cursor(Node* node=0): node_(node) {}
    const T& operator*() const { assert(node_); return node_->car(); }
    Cursor& operator++() { assert(node_); node_ = node_->cdr(); return *this; }
    Cursor operator++(int) { Cursor tmp(*this); ++*this; return tmp; }
    bool operator==(const Cursor& c) const { return node_ == c.node_; }
  private:
    Node* node_;
```

```
  };
private:
  Node(const T& t, Node* cdr): value_(t), cdr_(cdr) {}
  T value_;
  Node* cdr_;
};
```

Thus the **`Node<T>::Cursor`** class serves as an abstraction of the container for use by the algorithm.

```
extern Node<int>* list;
find(Node<int>::Cursor(list), Node<int>::Cursor(), 20);
```

In general, applying generic programming to container types and associated algorithms naturally leads to the introduction of so-called *iterator* classes that abstract from a particular container type. In this way, an algorithm only deals with iterators, independently of the underlying container structure, as illustrated in Figure 7.1. Intuitively, an iterator is an abstract pointer into the container. It typically supports

Figure 7.1: Iterators abstract from containers.

operations such as **`*`** and **`->`** (dereference), **`++`** (advance), assignment, etc.

```
template <typename T>
class C { // container class
public:
  // return iterator pointing to beginning of the container
  CIterator begin();
  // return iterator pointing past the end of the container
  CIterator end();
// ...
};
template <typename T>
class CIterator { // iterator pointing into Container
public:
  T operator*();
  T* operator->();
  CIterator operator++();
// ...
};
```

This is illustrated in Figure 7.2.

Depending on the container type, an iterator for such a container may support more powerful navigation operations. E.g. a singly linked list efficiently supports the increment (advance) operation but not decrement (go back). Doubly linked lists support both increment and decrement. More powerful containers such as arrays and vectors

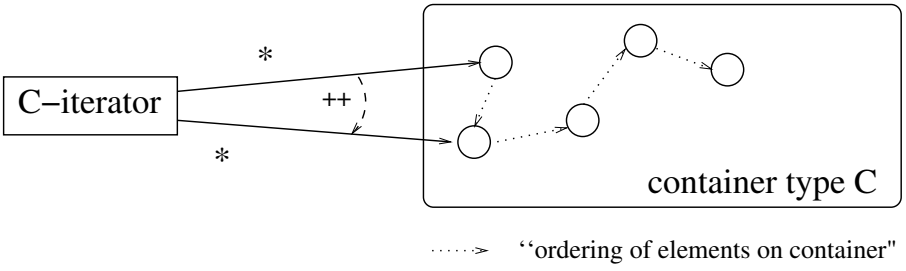

Figure 7.2: Iterators as abstract pointers.

additionally support random access using integral indices. On the other hand, an algorithm may require certain capabilities of the underlying iterator in order to function properly.

To model this variety, the STL distinguishes five kinds of iterators (we use **`it`** as an example iterator object):

- *Input iterators* allow just read-only access on the result of the dereference operator (**`x = *it; y = it->name`**). In addition the iterator can be advanced (**`++it`** and **`it++`**). Input iterators are very weak and mainly used to model access on an input stream. Nevertheless, an input iterator is enough to enable some useful algorithms, e.g. the **`find`** algorithm.

- *Output iterators* are the dual of input iterators; besides **`++it`** and **`it++`**, only write-only access to the result of the dereference operator (**`*it = x`**) is allowed. Output iterators are typically used to model writing to an output stream.

- *Forward iterators* combine the capabilities of input and output iterators: full access to the result of dereferencing (**`*it`**,**`it->name`**) as well as advancing in the container (**`++it`**,**`it++`**). Singly linked lists typically support forward iterators only.

- *Bidirectional iterators* improve upon forward iterators by providing a decrement (move back) capability (**`--it`** and **`it--`**). E.g. a doubly linked list implementation can be equipped with a bidirectional iterator.

- The most powerful iterator category contains *random access iterators*. Besides all the capabilities of bidirectional iterators, such iterators also support random access using an integral index (**`it+index`**, **`it[index]`**). E.g. ordinary **`T*`** pointers can serve as bidirectional iterators for the array type **`T[]`**.

The relationship of the requirements represented by the various iterator categories is illustrated in Figure 7.3 where arrows point to more powerful iterator categories.

Note that an iterator of a more powerful category than required by the algorithm may always be used with that algorithm. E.g. the **`find`** algorithm requires only an input

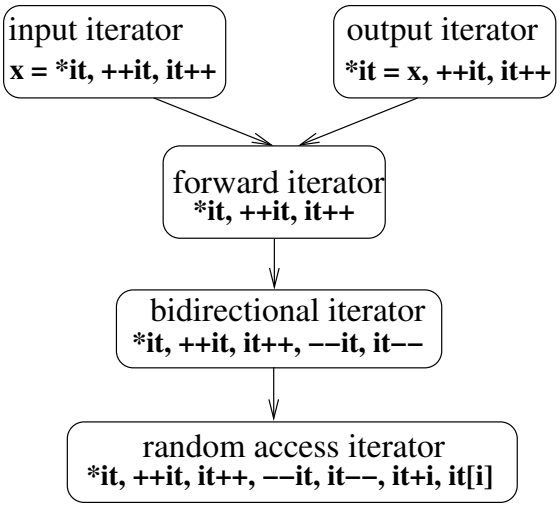

Figure 7.3: Iterator categories.

iterator but, since random access iterators subsume input iterators, it works as well with random access iterators, e.g. pointers into an array.

7.2.1 Types Associated with an Iterator

Consider the following algorithm which is supposed to return the sum of the elements in a range [*first . . . *last[of input iterators.

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

Unfortunately, the compiler is unable to deduce the value of the template type parameter **T** in a call which has not been explicitly instantiated.

```
extern Node<int>* l;
typedef Node<int>::Cursor  l_iterator;
sum(l_iterator(l), l_iterator()); // error
sum<l_iterator, int>(l_iterator(l), l_iterator()); // ok
```

The reason is that **T** is only used as a return type (and for a local variable), which is not taken into account by the template argument deduction procedure.

One way to get around this problem is to smuggle a **T&** parameter into **sum**'s signature.

```
template <class InputIterator, class T>
T
sum(InputIterator first, InputIterator last, T& result) {
if (first==last)
  return result;
do
  result += *first++;
while (first!=last);
return result;
}
```

Now code like the following will work without problems.

```
extern Node<int>* l;
int s(0);
sum(l_iterator(l), l_iterator(), s);
```

But this solution forces the algorithm designer to always have a **T** function argument if the return type is a template type parameter **T**. Also, the type parameter **T** should really be redundant since **T** is determined by **InputIterator**; specifically, **T** is the type of the object that an **InputIterator** refers to.

Thus a better solution demands that an iterator type **I** always defines a type **I::value_type** as the type of the objects that an **I** iterator points to, i.e. the "bare" version[6] of the return type of **I::operator*()**.

```
template <typename T>
class Node { // Node* is a (too) simple linked list
public:
  // ...
  class Cursor {
  public:
    typedef T value_type; // type to which cursor refers
    Cursor(Node* node=0): node_(node) {}
    const T&  operator*() const { assert(node_); return node_->car(); }
    Cursor&  operator++() { assert(node_); node_ = node_->cdr(); return *this; }
    Cursor  operator++(int) { Cursor tmp(*this); ++*this; return tmp; }
    bool  operator==(const Cursor& c) const { return node_ == c.node_; }
  private:
    Node*  node_;
  };
private:
  // ...
};
```

We can then rewrite the original **sum** algorithm of page 143 with only one template parameter.

[6] If **I::operator*()** returns **T&**, **I::value_type** should be **T**.

```
template <class InputIterator>
typename InputIterator::value_type
sum(InputIterator first, InputIterator last) {
assert(first!=last);
typename InputIterator::value_type result(*first++);
while (first!=last)
  result += *first++;
return result;
}
```

Note the use of the keyword **typename** in front of both uses of **InputIterator::value_type**. This is necessary to tell the compiler that the following name does refer to a type, which it would otherwise be unable to infer while parsing the template definition. The compiler can now easily deduce the single template type parameter from the function call arguments.

```
extern Node<int>* l;
typedef Node<int>::Cursor l_iterator;
sum(l_iterator(l), l_iterator()); // ok
```

There are some other types that are conventionally associated with an iterator type **I**.

- **I::value_type** is the type of the object an **I**-iterator points to. It is useful, e.g. to use as the type of a local variable in an algorithm.
- **I::difference_type** is an integral type large enough to hold the "distance" between two iterators (e.g. the number of times **I::operator++** must be executed to reach the other iterator). **I::difference_type** is almost always **ptrdiff_t**, which can hold the difference between pointers but for special applications (e.g. input iterators on terabyte files), this may not be large enough.
- **I::reference** is the return type of **I::operator*()**. Usually **I::reference** is **I::value_type&**. It is useful e.g. as a return type for a search algorithm.
- **I::pointer** is the type of **&I::reference**, which is also the return type of **I::operator->()**.
- **I::iterator_category** is a type that symbolizes the kind of iterator: input, output, forward, bidirectional or random access.

The value of **I::iterator_category** should be one of the following predefined empty iterator tag classes[7] (which can be found in the *iterator* header file).

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag : public input_iterator_tag {};
```

[7] Note the use of inheritance for some of these classes, see Chapter 8.

```
struct bidirectional_iterator_tag : public forward_iterator_tag {};
struct random_access_iterator_tag : public bidirectional_iterator_tag {};
```

Section 7.2.3 will illustrate the use of **`I::iterator_category`**.

When defining a new iterator, an easy way to define the associated types is by inheriting[8] from the STL class template **`iterator`**.

```
template <class Category, class T, class Distance = ptrdiff_t,
          class Pointer = T*, class Reference = T&>
struct iterator {
  typedef Category   iterator_category;
  typedef T          value_type;
  typedef Distance   difference_type;
  typedef Pointer    pointer;
  typedef Reference  reference;
};
```

E.g. **`Node<T>::Cursor`** can be defined as shown below.

```
template <typename T>
class Node { // Node* is a (too) simple linked list
public:
  // ...
  class Cursor: public iterator<forward_iterator_tag, T> {
  public: // value_type etc inherited from iterator
    Cursor(Node* node=0): node_(node) {}
    const T& operator*() const { assert(node_); return node_->car(); }
    Cursor& operator++() { assert(node_); node_ = node_->cdr(); return *this; }
    Cursor operator++(int) { Cursor tmp(*this); ++*this; return tmp; }
    bool operator==(const Cursor& c) const { return node_ == c.node_; }
  private:
    Node* node_;
  };
private:
  // ...
};
```

7.2.2 Iterator Traits

Defining types such as **`I::value_type`** associated with an iterator type **`I`** cannot be done for all iterators. E.g. a pointer type **`T*`** is an iterator type but, since it is not a class type, it is impossible to define **`T*::value_type`**.

Consequently, with the implementation of **`sum`** on page 144, the following will not compile because **`int*::value_type`** does not exist.

```
extern int[SIZE]  a;
sum(a, a+SIZE);
```

[8] See Chapter 8.

What we need is a compile-time function which, given an iterator type **I**, returns a type **T** that can be used as a **value_type** for **I**, i.e. we need a type constructor.

This can be achieved by using class templates and partial specialization[9].

The **iterator_traits** class template simply delegates the associated type definitions to the iterator type parameter class.

```
template <class Iterator>
struct iterator_traits {
  typedef typename Iterator::iterator_category iterator_category;
  typedef typename Iterator::value_type        value_type;
  typedef typename Iterator::difference_type   difference_type;
  typedef typename Iterator::pointer           pointer;
  typedef typename Iterator::reference         reference;
};
```

For pointer types, we use a partial specialization.

```
template <class T>
struct iterator_traits<T*> {
  typedef random_access_iterator_tag iterator_category;
  typedef T                          value_type;
  typedef ptrdiff_t                  difference_type;
  typedef T*                         pointer;
  typedef T&                         reference;
};
```

Thus, in the implementation of a general algorithm with an iterator type parameter **I**, we use **iterator_traits<I>::value_type** instead of **I::value_type**.

```
template <class InputIterator>
typename iterator_traits<InputIterator>::value_type
sum(InputIterator first, InputIterator last) {
assert(first!=last);
typename iterator_traits<InputIterator>::value_type result(*first++);
while (first!=last)
  result += *first++;
return result;
}
```

The example will now be processed correctly; for the call `sum(a, a+SIZE)`, the compiler infers that **a** is **int*** and, consequently, **InputIterator=int*** in the definition of **sum**. Using the specialized version of **iterator_traits**, it then discovers that the return type of **sum** is **iterator_traits<int*>::value_type**, i.e. **int**.

[9] For compilers, such as egcs-2.91.66, that do not support partial specialization, less elegant workarounds are available, see [Aus98].

7.2.3 Dispatching on the Iterator Category

Some (auxiliary) algorithms may be implemented in more efficient ways, depending on the capabilities of the underlying iterator **I**. We can use the **I::iterator_category** type to create overloaded function templates such that the compiler will automatically select the most efficient implementation.

```
template <class InputIterator>
inline void // version for input iterators and forward iterators
advance(InputIterator& i,
  typename iterator_traits<InputIterator>::difference_type n,
  input_iterator_tag) {
for (; n>0 ; --n)
  ++i;
}

template <class BidirectionalIterator>
inline void // version for bidirectional iterator
advance(BidirectionalIterator& i,
  typename iterator_traits<BidirectionalIterator>::difference_type n,
  bidirectional_iterator_tag) {
if (n>=0)
  for (; n>0 ; --n) ++i;
else
  for (; n<0 ; ++n) --i;
}

template <class RandomAccessIterator>
inline void // version for random access iterators
advance(RandomAccessIterator& i,
  typename iterator_traits<RandomAccessIterator>::difference_type n,
  random_access_iterator_tag) {
i += n;
}

template <class InputIterator>// the general version, it
inline void // dispatches to a more specialized one depending on the iterator kind
advance(InputIterator& i, typename iterator_traits<InputIterator>::difference_type n) {
advance(i, n, typename iterator_traits<InputIterator>::iterator_category());
}
```

In the above implementation, the main function template (we abbreviate **InputIterator** as **I**)

```
advance(I& i, typename iterator_traits<I>::difference_type n)
```

calls an overloaded function template of the same name but with an extra argument object of type **iterator_traits<I>::iterator_category**:

```
advance(i, n, typename iterator_traits<I>::iterator_category())
```

Note the use of the default constructor for the creation of the third argument.

Depending on the category, the compiler will select the appropriate template: if the extra argument is of type **input_iterator_tag**, the function simply calls **I::**

operator++() an appropriate number of times. If **I::iterator_category** is **bidirectional_iterator_tag**, the distance to advance may be negative, in which case, **I::operator--()** is used. Finally, if **I** is a random access iterator, arithmetic is used to obtain a much faster implementation.

The compiler processes a call such as

```
extern int* a;
advance(a, 10);
```

as follows: from the type **int*** of the first parameter, **I** is identified as **int***. Using the pointer specialization of **iterator_traits**, the type of the third argument of the call to **advance** is determined to be **random_access_iterator_tag**. Overload resolution for the call to

```
advance(int*, int, random_access_iterator_tag)
```

leads to the correct instantiation. Since all functions are inline, the compiler may reduce the call to

```
a += 10;
```

7.3 Stream Iterators

Most algorithms have an interface in terms of iterators. Hence, if we want to apply an algorithm to a container type, this type should have an iterator interface available. Stream iterators provide such an interface for input and output streams.

7.3.1 Input Stream Iterator

An **istream_iterator<T> it** makes an input stream look like a sequence of **T** objects which can be accessed using ***it**. The increment operator **++it** advances the current position in the stream. Since it is not possible to assign to ***it**, **istream_iterator<T>::iterator_category** is **input_iterator_tag**.

Here is a possible implementation[10]

```
template <class T, class Distance = ptrdiff_t>
class istream_iterator: public iterator<input_iterator_tag, T, Distance> {
public:
  istream_iterator(): stream(&cin), ok(false) {} // serves as ''past-the-end'' iterator
  istream_iterator(istream& s): stream(&s) { read(); }
  const T& operator*() const { return value; }
  istream_iterator& operator++() { read(); return *this; }
  istream_iterator operator++(int)        { istream_iterator tmp = *this; read(); return tmp; }
  friend bool
    operator==<T, Distance>(const istream_iterator& x, const istream_iterator& y);
private:
  istream* stream;
  T value;
```

[10] In a real implementation, the private area would be protected (Chapter 8) to enable the definition of derived classes.

```
  bool ok;
  void read() {
    ok = (*stream) ? true : false;
    if (ok)
      *stream >> value;
    ok = (*stream) ? true : false;
    }
};

template <class T, class Distance>
inline bool
operator==(const istream_iterator<T, Distance>& x, const istream_iterator<T, Distance>& y) {
return x.stream == y.stream && x.ok == y.ok || x.ok == false && y.ok == false;
}
```

The default constructor constructs an "end of stream" object **`eos`** which can serve as an "end of range" value in an algorithm; the end of the input stream is reached when **`it == eos`**.

Note also the declaration of the friend function **`operator==`** as a function template instantiation.

The following example uses a class **`Word`**, representing strings of alphabetic characters only.

```
#ifndef WORD_H
#define WORD_H
#include <string>
#include <iostream>
#include <ctype.h> // for isalpha()

class Word {
public:
  Word(const string& s=string("")): word_(s) {}
  const string&    word() const { return word_; }
  friend istream&
    operator>>(istream&, Word&);
  friend ostream&
    operator<<(ostream& os, const Word& w) { return os << w.word(); }
  bool  operator<(const Word& w) const { return word_ < w.word(); }
  bool  operator==(const Word& w) const { return word_ == w.word(); }
private:
  string    word_;
};

inline istream&
operator>>(istream& is, Word& w) {
char c;
do // skip until first alphabetic char
  c = is.get();
while ((is) && (!isalpha(c)));
if (!is)
  return is;
w.word_ = ""; // initialize word_ and then read consecutive letters
while ((is) && isalpha(c)) {
  w.word_ += tolower(c); // append c
```

```
  c = is.get();
  }
return is;
}
#endif
```

The implementation is straightforward; only the input operator **operator>>** needs some care. We use the low-level character-by character member function[11] **istream::get()** which, unlike e.g. **operator>>(istream&,string&)**, does not skip *white space* (**'\t'**, **' '** and **'\n'**). We also use **string::operator+(char)** which appends a character to the end of a string.

The example program looks for the occurrence of a certain word in the input stream.

```
#include <algorithm>
#include <iterator>
#include "word.h"

int
main(int argc, char* argv[]) {
if (argc<2)
  return 1;
Word word(argv[1]);
istream_iterator<Word>    input(cin);
istream_iterator<Word>    end;
istream_iterator<Word>    found(find(input, end,word));
if (found!=end)
  cout << *found << endl;
}
```

The program uses the **find** algorithm on the interval $[input \ldots end[$ to locate a position in the file where **word** can be found.

7.3.2 Output Stream Iterator

Just as **istream_iterator<T>** makes an input stream iterator look like a retrieve-only container, an **ostream_iterator<T> it** makes an output stream look like a write-only container.

```
template <class T>
class ostream_iterator: public iterator<output_iterator_tag, void, void, void> {
public:
  ostream_iterator(ostream& s): stream(&s), separator_string(0) {}
  ostream_iterator(ostream& s, const char* c): stream(&s), separator_string(c)        {}
  ostream_iterator& operator=(const T& value) {
    *stream << value;
    if (separator_string)
      *stream << separator_string;
    return *this;
    }
```

[11] See Chapter 10.

```
  ostream_iterator& operator*() { return *this; }
  ostream_iterator& operator++() { return *this; }
  ostream_iterator& operator++(int) { return *this; }
private:
  ostream* stream;
  const char* separator_string;
};
```

Since **ostream_iterator<T>** is an output iterator, the only operations that must be implemented are **++it** and ***it = t** where **t** is of type **T**. In particular, ***it** cannot be accessed, only written to. Hence iterator types such as **value_type** do not make sense and the implementation declares them as **void**. The only member function that really does any work is the assignment operation; it writes its argument followed by the separator string. The implementation of **operator*** and **operator++** only serve to make expressions like ***it++ = t** behave as expected.

The following example uses the **copy** algorithm[12].

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

We can then use this algorithm and an output stream iterator to write the contents of a container to a file.

```
double data[SIZE];
copy(data, data+SIZE, ostream_iterator<double>(cout,"\n"));
```

7.4 STL Containers

The standard library supports several types of containers with different capabilities and performance characteristics. Here we briefly discuss only the most important containers and their most common member functions. For a more complete discussion, see e.g. [Aus98] or the SGI web site on STL at **http://www.sgi.com/tech/stl**.

We distinguish three kinds of containers in STL:

- *Sequence containers* represent a sequence of elements. E.g. **vector** and **list** belong in this category.
- *Associative containers* represent a finite mapping $Key \rightarrow Value$ between two types. Examples include **map** and **set**.

[12] The real implementation dispatches on the iterator types such that, e.g. for pointers into an array, fast memory move instructions are used, see page 169.

- *Adaptors* are class templates that put a new interface (ADT) on top of an existing underlying container implementation. E.g. a **`stack`** can be implemented based on any sequence container.

7.4.1 Pair

This is an auxiliary container. A pair is simply an ordered pair of objects that may be of different types.

```
template <class T1, class T2>
struct pair {
  typedef T1 first_type;
  typedef T2 second_type;
  T1 first;
  T2 second;
  pair() : first(T1()), second(T2()) {}
  pair(const T1& a, const T2& b) : first(a), second(b) {}
};
template <class T1, class T2>
inline bool
operator==(const pair<T1, T2>& x, const pair<T1, T2>& y) {
return x.first == y.first && x.second == y.second;
}
template <class T1, class T2>
inline bool
operator<(const pair<T1, T2>& x, const pair<T1, T2>& y) {
return x.first < y.first || (!(y.first < x.first) && x.second < y.second);
}
template <class T1, class T2>
inline pair<T1, T2> make_pair(const T1& x, const T2& y) {
return pair<T1, T2>(x, y);
}
```

As can be seen from the definition, pairs are compared "lexicographically". Also note the **`make_pair`** convenience function; it allows us to write e.g.

```
f(make_pair(a, b));
```

rather than the more cumbersome

```
f(pair<type_of_A,type_of_B>(a, b));
```

7.4.2 List

A **`list`** represents an extendible list of objects of the same type. Lists support efficient $\mathcal{O}(1)$ addition and deletion, at arbitrary positions in the sequence. Moreover, unlike with vectors, list updates do not invalidate existing iterators that refer to list elements (except the ones that are explicitly deleted). The underlying implementation is often a doubly linked list.

```
template <class T, class Alloc = alloc>
class list {
```

```
public:
  typedef T value_type;
  typedef value_type* pointer;
  typedef value_type& reference;
  typedef const value_type& const_reference;
  typedef size_t size_type;
  typedef ptrdiff_t difference_type;
  typedef ... iterator;
  typedef ... const_iterator;

  // constructors, destructor, assignment
  list(); // create empty list
  list(size_type n, const T& value); // create list containing n copies of value
  explicit list(size_type n); // create list containing n copies of T::T()
  template <class InputIterator>
    list(InputIterator first, InputIterator last); // new list contains [*first .. *last[
  list(const list<T, Alloc>& x); // copy constructor copies the whole list
  ~list();
  list<T, Alloc>& operator=(const list<T, Alloc>& x);
  // accessor functions
  iterator begin(); // return reference to first element or end()
  const_iterator begin() const;
  iterator end(); // return reference past the end
  const_iterator end() const;
  bool empty() const; // true if list has no elements
  size_type size() const; // returns number of elements in the list
  size_type max_size() const; // a *huge* number
  reference front(); // reference to first element of the list, if !empty()
  const_reference front() const;
  reference back(); // reference to last element of the list, if !empty()
  const_reference back() const;
  // inserting new elements
  iterator insert(iterator position, const T& x); // insert x before position
  iterator insert(iterator position); // insert T::T()
  template <class InputIterator> void // insert [*first .. *last[ before position
    insert(iterator position, InputIterator first, InputIterator last);
  void insert(iterator pos, size_type n, const T& x); // insert n copies of x
  void push_front(const T& x); // prepend x, i.e. insert(begin(),x)
  void push_back(const T& x); //      append x, i.e. insert(end(),x)
  // removing elements
  void erase(iterator position); // remove element at position
  void erase(iterator first, iterator last); // remove [first..last[
  void clear(); // remove all elements
  void pop_front(); // erase first element
  void pop_back(); // erase last element
  // transfering elements from another list into this list
  void splice(iterator position, list& x); // move all elements from x to before position
  void splice(iterator position, list& x, iterator i); // move *i from x to before position
  void splice(iterator position, list&, iterator first, iterator last); // move [*first..*last[ ..
  // other operations
  void remove(const T& value); // search and remove
  void unique(); // remove all but the first of all groups of consecutive equal elements
  void merge(list& x); // merge ordered list x into this ordered list
  void reverse(); // reverse the order of the elements
  void sort();
  // generalizations
  template <class Predicate> void
```

```
    remove_if(Predicate p); // remove x such that p(*x) is true
  template <class BinaryPredicate> void
    unique(BinaryPredicate p); // remove all x+1 such that p(*x, *(x+1))
  template <class StrictWeakOrdering> void
    merge(list&, StrictWeakOrdering comp); // like merge but comp(x,y) instead of x<y
  template <class StrictWeakOrdering> void
    sort(StrictWeakOrdering comp); // like sort but use comp(x,y) instead of x<y
};

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

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

The second template parameter determines the type of allocator object that will be used for the internal memory management of the list. Note that two bidirectional iterators are supported: **`list<T>::iterator`** and **`list<T>::const_iterator`**. Modifications to a list element are not possible using the latter since e.g. **`const_iterator::operator*`** returns a **`const T&`**.

The following example reads all words from the input stream, sorts them and removes duplicates, and then writes them out to the output stream.

```
#include <iterator>
#include <list>
#include <string>
#include <algorithm>
#include <iostream>
#include "word.h"

int
main(int, const char**) {
istream_iterator<Word>    input(cin);
istream_iterator<Word>    end_of_input;
list<Word>  words(input,end_of_input); // reads words from cin into a list
words.sort();
words.unique(); // after each element, remove subsequent duplicates, e.g. a a b a becomes a b a
ostream_iterator<Word>    output(cout, "\n");
copy(words.begin(), words.end(), output); // copy list to cout
}
```

A singly linked list container **`slist`** also exists, see e.g. [Aus98].

7.4.3 Vector

A **`vector<T>`** is a sequence container that supports efficient $\mathcal{O}(1)$ random access using the subscript operator. Insertion at the end is efficient ($\mathcal{O}(1)$) but insertion at the beginning or in the middle is expensive ($\mathcal{O}(n)$). The underlying implementation is

an array which may be reallocated when needed. Thus, iterators do not remain valid under insertion or erasure of elements.

```
template <class T, class Alloc = alloc>
class vector {
public:
  typedef T value_type;
  typedef value_type* pointer;
  typedef value_type* iterator;
  typedef const value_type* const_iterator;
  typedef value_type& reference;
  typedef const value_type& const_reference;
  typedef size_t size_type;
  typedef ptrdiff_t difference_type;
public:
  // constructors, destructor, assignment
  vector();
  vector(size_type n, const T& value); // vector with n elements = value
  explicit vector(size_type n); // vector with n elements = T::T()
  vector(const vector& x); // copy constructor
  template <class InputIterator> // vector containing [*first .. *last[
    vector(InputIterator first, InputIterator last);
  ~vector();
  vector& operator=(const vector& x); // assignment
  // accessors
  iterator begin();
  const_iterator begin() const;
  iterator end();
  const_iterator end() const;
  size_type size() const;
  size_type max_size() const; // a huge number
  size_type capacity() const; // current capacity, size() <= capacity()
  bool empty() const;
  reference operator[](size_type n); // random access
  const_reference operator[](size_type n) const; // random access
  reference at(size_type n); // checking version of operator[ ]
  const_reference at(size_type n) const; // checking version of operator[ ]
  reference front(); // reference to first element if !empty()
  const_reference front() const;
  reference back(); // reference to last element if !empty()
  const_reference back() const;
  // inserting elements
  void push_back(const T& x); // append
  iterator insert(iterator position, const T& x = T::T()); // insert new element before position
  template <class InputIterator> // insert [*first .. *last[ before position
    void insert(iterator position, InputIterator first, InputIterator last);
  void insert (iterator pos, size_type n, const T& x); // insert n copies of x before pos
  // removing elements
  void pop_back(); // remove last element
  void erase(iterator position); // remove element at position
  void erase(iterator first, iterator last); // erase [first .. last[
  void resize(size_type new_size, const T& x); // resize by erasing (at end) or appending x's
  void clear(); // erase all elements
  // other
  void reserve(size_type n); // ensure n <= capacity()
};
template <class T, class Alloc>
```

```
inline bool
operator==(const vector<T, Alloc>& x, const vector<T, Alloc>& y);

template <class T, class Alloc>
inline bool // lexicographical comparison of elements
operator<(const vector<T, Alloc>& x, const vector<T, Alloc>& y);
```

Vector supports two random access iterators: **`vector<T>::iterator`** and **`vector<T>::const_iterator`**. Modifications through a constant iterator are not possible.

The following example builds a hash table with separate chaining out of a file of words (strings). The hash table is implemented as a preallocated vector of lists of strings. When inserting a string, its hash address (obtained using division hashing of the product of all the string's symbols) is used as an index in the vector. The string is then appended to the list at that position in the vector.

```
#include <string>
#include <vector>
#include <list>

unsigned int
scramble(string s) {
unsigned int result=1;
for (unsigned int i=0;(i<s.size());++i)
  result *= s[i];
return result;
}

class Dictionary {
public:
  Dictionary(unsigned int size): size_(size), words_(size) {}
  bool find(const string& s) {
    list<string>&  l(words_[hash(s)]);
    return ::find(l.begin(), l.end(), s) != l.end();
    }
  void insert(const string& s) { if (!find(s)) words_[hash(s)].push_back(s); }
  static unsigned int hash(const string& s) { return scramble(s)%size_; }
private:
  unsigned int  size_;
  vector<list<string> >  words_;
};
```

Note the use of **`::find()`** in the implementation of **`Dictionary::find`**; we must explicitly refer to the **`find`** algorithm in global scope because otherwise the compiler would interpret it as referring to **`Dictionary::find`**.

7.4.4 Map

A **`map<Key,Value>`** is an associative container that associates objects of type **`Key`** with objects of type **`Value`**. Moreover, keys are unique, i.e. at most one **`Value`**

object may be associated with a particular **`Key`** value[13]. Mathematically, a map simply represents a finite partial function.

$$\mathbf{map}\langle Key, Value\rangle \;\; m : Key \rightarrow Value$$

Map iterators are bidirectional; traversal is in key order. The **`value_type`** of an iterator is **`pair<const Key,Value>`**. The implementation is usually based on a balanced (red–black) tree. This explains why the first component of the value type of an iterator is **`const Key`**; modifying the **`Key`** of an element in the map would wreak havoc on the internal tree structure.

```
template <class Key, class Value, class Compare = less<Key>, class Alloc = alloc>
class map {
public:
  // typedefs:
  typedef .. pointer;
  typedef .. reference;
  typedef .. const_reference;
  typedef .. iterator; // bidirectional, use operator[ ] or find() for ''random'' access
  typedef .. const_iterator;
  typedef .. size_type;
  typedef .. difference_type;
  // constructors, assignment
  map();
  explicit map(const Compare& comp); // with non-default Key Compare function object
  template <class InputIterator>
    map(InputIterator first, InputIterator last); // insert [*first .. *last[
  template <class InputIterator>
    map(InputIterator first, InputIterator last, const Compare& comp);
  map(const map& x); // copy constructor
  map& operator=(const map&); // assignment
  // accessors:
  iterator begin(); // return iterator referring to first element (in Key order)
  const_iterator begin() const;
  iterator end(); // ''past the end''
  const_iterator end() const;
  bool empty() const;
  size_type size() const; // number of elements in the mape
  size_type max_size();
  // insert/erase:
  pair<iterator, bool> insert(const pair<Key, Value>& element); // see text
  template <class InputIterator>
    void insert(InputIterator first, InputIterator last);
  void erase(iterator position); // erase element at position
  size_type erase(const Key& k); // erase all k's, return # of elements erased
  void erase(iterator first, iterator last); // erase elements in [first..last[
  void clear(); // erase all elements in map
  // map operations:
  Value& operator[](const Key& k); // random access: return value associated with key, creates
      // entry (with default Value) if not found
  iterator find(const Key& k); // find value with key, return end() if not found
  const_iterator find(const Key& k) const;
  size_type count(const Key& k) const; // number of elements with Key k
```

[13]There is also a **`multi_map`** container that does not have this restriction, see Section 7.4.6.

```
  iterator lower_bound(const Key& k); // first element with key >= k
  const_iterator lower_bound(const Key& k) const;
  iterator upper_bound(const Key& k); // first element with key > k
  const_iterator upper_bound(const Key& k) const;
  pair<iterator, iterator> equal_range(const Key& k); // range of all elements with key = k,
      // more useful for multi_map
  pair<const_iterator, const_iterator> equal_range(const Key& k) const;
  friend bool operator==(const map&, const map&);
  friend bool operator<(const map&, const map&);
};
```

A **map<Key,Value>** map supports two member functions for efficient $\mathcal{O}(\log n)$ access to an element, based on a particular **Key** value:

- **map<Key,Value>::find(const Key& k)** returns an iterator (**end()**, if not found) referring to the unique element, if present, containing the key value **k**.
- **map<Key,Value>::operator[](const Key& k)** returns a reference to the element with key value **k**. If no such element exists, a new **make_pair(k,Value::Value())** is inserted.

The second method is convenient for updating a map using an intuitive "associative array" syntax: **m[k] = value**. If we only want to know whether an element with a certain key **k** is present, the member function **map<Key,Value>::count(const Key& k)** is the most efficient choice.

The **insert(const pair<Key,Value>& element)** member function returns a **pair<iterator,bool>** pair. The **bool** component is true iff the **element** was successfully inserted; the **iterator** component always refers to an element with a key value equal to the parameter **element**'s key. Thus, if the map already contained an element with the parameter key, the first component will refer to this element.

The following example implements *sparse matrices*, i.e. two-dimensional arrays where most entries have a default value. For large matrices, the obvious implementation, e.g. one using vectors of vectors, wastes too much memory, especially if only a few entries have a non-default value. In our implementation, only elements with non-default values are explicitly stored (in a **map**), resulting in substantial space saving, at the cost of slightly more expensive $\mathcal{O}(\log n)$ instead of $\mathcal{O}(1)$ access.

```
template<typename Key1, typename Key2, typename Value>
class SparseMatrix {
public:
  class Element; // forward declaration
  class Row { // a row is a [Key2->Value] map
  friend class Element;
  public:
   Row(SparseMatrix& matrix): matrix_(matrix) {} // remember matrix for getting default value
   Element  operator[](const Key2& k2) { return Element(*this, k2); }
```

```
  private:
   Row& operator=(const Row&); // forbid assignment
   SparseMatrix&   matrix_;
   map<Key2, Value> columns_; // one element for every non-default element in row
  };
  class Element {
  public:
    Element(Row& r, Key2 k2): row_(r), key2_(k2) {}
    void operator=(const Value& val) {     // e.g. a[k1][k2] = val
      if (val==matrix().default_value())          // val is default, erase possible k2 entry from row
        row_.columns_.erase(key2_);
      else
        row_.columns_[key2_] = val;      // non-default val: store in row[k2]
      }
    operator Value() { // return value of a[k1][k2]
      if (row_.columns_.count(key2_)>0) // a[k1][k2] explicitely stored: retrieve it
        return row_.columns_[key2_];
      else // a[k1][k2] not stored: return default value
        return matrix().default_value();
      }
  private:
    SparseMatrix& matrix() const { return row_.matrix_; }
    Row& row_; // e.g. a[k1][k2] returns element refering to a[k1] row, key2=k2
    Key2  key2_;
  };
  SparseMatrix(const Value& default_value): default_(default_value) {}
  const Value&   default_value() const { return default_; }
  Row& operator[](const Key1& k1) {
    // return reference to row from rows_ (possibly after inserting it)
    return (*((rows_.insert(make_pair(k1, Row(*this)))).first)).second;
    }
private:
  Value   default_;
  map<Key1, Row> rows_;
};
```

The above implementation stores a matrix with index types **`Key1`** and **`Key2`** as a **`map<Key1,Row>`** where **`Row`** is another **`map<Key2,Value>`**. A reference **`m[k1][k2]`** to an element of the matrix does not immediately return a **`Value`**. Rather, **`m[k1][k2]`** returns an **`Element`** object that essentially consists of the pair **`<row,k2>`** where **`row`** refers to the **`k1`**-row map. The element corresponding to **`m[k1][k2]`** is obtained in two steps:

- First **`m.operator[](k1)`** is invoked, which results in a (reference to a) **`Row`**, say **`r`**.
- Then **`r.operator[](k2)`** is evaluated, which results in the element **`<row,k2>`**.

An **`Element`** can be regarded as a "virtual" matrix entry that can be further evaluated to an actual **`Value`**, or it can be assigned to.

- The former is done "on demand" thanks to the explicit converter function **`Element::operator Value()`** which, when applied to a **`<row,k2>`** element, will access the **`row`** map to find out if the element has a non-default value; if it does not, the default value is returned. Thus expressions like

```
SparseMatrix<int,int,double> m(0);
sqrt(m[100][100]);
```

  will be evaluated as expected; the compiler will attempt to convert the **`Element`**, say **`e`**, resulting from **`m[100][100]`** to match **`sqrt`**'s parameter type **`double`**, thus invoking **`e.operator double()`**.

- A sparse matrix can be updated using the normal syntax; in

```
SparseMatrix<int, int, double> m(0);
m[99][1004] = 3.14;
```

  `m[99][1004]` will evaluate to an **`Element e`** for which the assignment operation **`e.operator=(3.14)`** will be invoked. Depending on the value to be assigned, an explicit element may be stored or erased (if the default value is assigned) in the **`e`**'s row map.

7.4.5 Set

A **`set<Key>`** is like a **`map<Key,Key>`**, i.e. a mapping from the set element type **`Key`** to itself. Mathematically, a set represents a subset of the identity function

$$\mathbf{set}\langle Key\rangle \;\; s : Key \rightarrow Key$$

where $s(x) = x$ for all x where s is defined. Thus keys are unique, i.e. at most one element in a set may be associated with a particular **`Key`** value[14]. Set iterators are bidirectional; traversal is in key order. The **`value_type`** of an iterator is simply **`Key`** but, since **`set<T>::iterator`** is constant, it is not possible to modify a set element through an iterator; i.e. `*it = other_key` will not compile. As for maps, the implementation of sets is usually based on a balanced (red–black) tree. This explains why iterators are constant; modifying an element in the set would upset the internal tree structure.

The interface of **`set<Key>`** is a subset of the one provided by **`map`**.

```
template <class Key, class Compare = less<Key>, class Alloc = alloc>
class set {
public:
  // typedefs:
public:
  typedef ... pointer;
  typedef ... const_reference;
  typedef const_reference reference; // all iterators and references are constant
  typedef ... const_iterator;
  typedef const_iterator iterator;
```

[14] There is also a **`multi_set`** container that does not have this restriction, see Section 7.4.6.

```
  typedef ... size_type;
  typedef ... difference_type;
  // constructor
  set();
  explicit set(const Compare& comp);     // bool Compare::Compare(const Key&, const Key&);
  template <class InputIterator>
    set(InputIterator first, InputIterator last); // fill with [*first .. *last[
  template <class InputIterator>
    set(InputIterator first, InputIterator last, const Compare& comp);
  set(const set& x); // copy constructor
  set& operator=(const set& x); // assignment
  // accessors:
  iterator begin() const; // note that only a const version exists
  iterator end() const;
  bool empty() const;
  size_type size() const; // number of elements in the set
  size_type max_size();
  // insert, erase;
  pair<iterator, bool> insert(const value_type& x); // if p = aset.insert(x) then
    // p.second = true iff x was newly inserted, p.second always refers to element with key x
  template <class InputIterator>
    void insert(InputIterator first, InputIterator last); // insert [*first .. *last[
  void erase(iterator position);
  size_type erase(const key_type& x);
  void erase(iterator first, iterator last); // erase [*first .. *last[
  void clear(); // remove all elements
  // set operations:
  iterator find(const key_type& x) const; // return end() or it where *it == x
  size_type count(const key_type& x) const; // return 1 iff x is in set, 0 otherwise
  iterator lower_bound(const key_type& x) const; // first element with key >= x
  iterator upper_bound(const key_type& x) const; // first element with key > x
  pair<iterator, iterator> equal_range(const key_type& x) const; // return
    // <i1, i2> such that *i == x for all i in      [i1 .. i2[
  friend bool operator==(const set&, const set&);
  friend bool operator<(const set&, const set&);
};
```

The following example uses sets to build a dictionary with a similar functionality as the one on page 157[15].

```
#include <string>
#include <set>

class Dictionary {
public:
  bool   find(const string& s) { return words_.count(s)>0; }
  void   insert(const string& s) { words_.insert(s); }
private:
  set<string>    words_;
};
```

[15] Building a 40 000-word dictionary using the set version is about 33% faster than the hashed version (with a hash table of size 998). Increasing the size of the hash table to 9980 makes the hashed version faster than the set version (by about 18%).

7.4.6 Other Containers

Besides **list**, **vector**, **map** and **set**, the STL also supports the following container types:

- A *deque* ("double-ended queue") container is similar to a vector (Section 7.4.3). The main differences are that a deque supports $\mathcal{O}(1)$ insertion at the end *and* in the beginning of the container.

- A *slist* ("singly linked list") container is smaller and faster than a list (Section 7.4.2). However, slists supports only forward iterators (vs. bidirectional iterators for lists).

- A *multimap* container is very similar to a map (Section 7.4.4). The main difference is that a multimap may contain several elements with the same key value.

- A *multiset* resembles a set (Section 7.4.5); the only difference is that it may contain several identical elements.

- Maps and sets support ordered (on the key) access through their iterators. In hash-based container types, this ordered access is sacrificed for more efficient (average $\mathcal{O}(1)$) retrieval of elements corresponding to a given key value. Although not required by the standard, most implementations provide hashed versions of all tree-based associative containers: *hash_map*, *hash_set*, *hash_multimap*, *hash_multiset*.

Complete information on these container types can be found elsewhere, e.g. in [Aus98], or on **http://www.sgi.com/tech/stl**.

7.4.7 Container Adaptors

A *container adaptor* is a template-based implementation of a container ADT for which the underlying "real" container is a template parameter. Hence an adaptor comes close to a pure definition of an abstract data type (Section 4.1), i.e. a definition that is independent of the underlying implementation[16].

The STL supplies adaptor classes for *stack*s, *queue*s and *priority_queue*. Here we only show the definition of **stack**.

[16] Pure definitions of ADTs can also be done using abstract classes (Section 8.2.2) but such an approach is intrusive on the implementation subclass. In contrast, an adaptor is generic in that it makes only minimal assumptions on the underlying implementation container, and the latter is completely independent of the ADT. Moreover, the generic solution, unlike a solution based on inheritance, does not carry a performance cost.

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

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

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

Note that the default implementation **`Sequence`** template parameter is *deque* but a *vector* or *list* can also be used.

```
stack<string>  deque_stack;
stack<string,vector<string> >  vector_stack;
stack<string,list<string> >  list_stack;
```

7.5 STL Algorithms

The STL defines approximately 75 algorithms, most of them dealing with containers through iterators. In this section we briefly summarize the available algorithms. For more information and an in-depth discussion about generic programming and the STL, we refer to [Aus98] or **`http://www.sgi.com/tech/stl`**.

In the listings below, we use the "big O" notation to indicate the complexity of the algorithm as a function on the size of its input. We use **`**`** to denote exponentiation; e.g. **`O(n**2)`** stands for $\mathcal{O}(n^2)$. All algorithms can be defined by including the *algorithm* header file, except for the numeric algorithms (page 173) for which the file *numeric* must be included.

7.5.1 Non-mutating Algorithms

These algorithms extract information from an iterator range.

Finding elements in a range

```
template <class InputIterator, class T>
InputIterator // return i where *i == value or last; O(n)
find(InputIterator first, InputIterator last, const T& value);

template <class InputIterator, class Predicate>
InputIterator // return i where pred(*i) or last; O(n)
find_if(InputIterator first, InputIterator last, Predicate pred);

template <class ForwardIterator>
ForwardIterator // return i where *i == *(i+1) or last; O(n)
adjacent_find(ForwardIterator first, ForwardIterator last);

template <class ForwardIterator, class BinaryPredicate>
ForwardIterator // return i where pred(*i,*(i+1)) or last; O(n)
adjacent_find(ForwardIterator first, ForwardIterator last, BinaryPredicate binary_pred);

template <class InputIterator, class ForwardIterator>
InputIterator // return first i in [first1..last1[ where *i in [*first2..*last2[ or last1; O(n**2)
find_first_of(InputIterator first1, InputIterator last1, ForwardIterator first2, ForwardIterator last2);

template <class InputIterator, class ForwardIterator, class BinaryPredicate>
InputIterator // return first i in [first1..last1[ where
              // comp(*i,*j), j in [*first2..*last2[ or last1; O(n**2)
find_first_of(InputIterator first1, InputIterator last1,
              ForwardIterator first2, ForwardIterator last2, BinaryPredicate comp);
```

Finding subranges

```
template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 // return i where [*i..*last1[ starts with [*first2..*last2[
                 // or last1; worst case O(n**2), average O(n)
search(ForwardIterator1 first1, ForwardIterator1 last1,
       ForwardIterator2 first2, ForwardIterator2 last2);

template <class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
ForwardIterator1 // like above but uses comp instead of ==; worst case O(n**2), average O(n)
search(ForwardIterator1 first1, ForwardIterator1 last1,
       ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate comp);

template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 // like search but starts at the back; i.e. return last i ..
         // O(n**2), average O(n) if all iterators are bidirectional
find_end(ForwardIterator1 first1, ForwardIterator1 last1,
         ForwardIterator2 first2, ForwardIterator2 last2);

template <class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
ForwardIterator1 // like search but starts at the back; i.e. return last i ..
                 // O(n**2), average O(n) if all iterators are bidirectional
find_end(ForwardIterator1 first1, ForwardIterator1 last1,
         ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate comp);
```

```
template <class ForwardIterator, class Integer, class T>
ForwardIterator // return i where for all j in [i..(i+count)[ *j == value or last; O(n)
search_n(ForwardIterator first, ForwardIterator last, Integer count, const T& value);

template <class ForwardIterator, class Integer, class T, class BinaryPredicate>
ForwardIterator // like above but use comp instead of ==; O(n)
search_n(ForwardIterator first, ForwardIterator last, Integer count,
         const T& value, BinaryPredicate comp);
```

Counting elements in a range

```
template <class InputIterator, class T, class Size>
void // set n to number of elements j in [first..last[ where *j == value; O(n)
count(InputIterator first, InputIterator last, const T& value, Size& n);

template <class InputIterator, class T>
typename iterator_traits<InputIterator>::difference_type // like above but return n; O(n)
count(InputIterator first, InputIterator last, const T& value);

template <class InputIterator, class Predicate, class Size>
void // set n to number of elements j in [first..last[ where pred(*j); O(n)
count_if(InputIterator first, InputIterator last, Predicate pred, Size& n);

template <class InputIterator, class Predicate>
typename iterator_traits<InputIterator>::difference_type // like above but return n; O(n)
count_if(InputIterator first, InputIterator last, Predicate pred);
```

Processing a range

The **`for_each`** algorithm resembles the **`map`** functional that is often found in languages such as Lisp. It applies its unary function (or unary function object) to each of the elements in the range.

```
template <class InputIterator, class Function>
Function // apply f(*i) for each i in [first..last[, return f; O(n)
for_each(InputIterator first, InputIterator last, Function f);
```

In order to facilitate the usage of member functions by algorithms such as **`for_each`**, the STL provides class templates for function objects that apply a given member function for a parameter target object.

```
template <class S, class T> // function object such that F(f)(x) applies x->*f()
class mem_fun_t: public unary_function<T*, S> {
public:
  explicit mem_fun_t(S (T::*pf)()) : f(pf) {}
  S operator()(T* p) const { return (p->*f)(); }
private:
  S (T::*f)();
};
```

There are actually eight variations of the above class template, each supporting a combination of the following features:

- the member function is **const** or not;
- the member function is called through a pointer to the target object or through a reference; and
- the member function takes an argument or not.

Luckily, one does not have to remember the names of the template classes; it suffices to use one of four overloaded helper functions that generate an appropriate instantiation:

- **mem_fun(R (T::*f)())** for member functions **R T::f()** that are called through a pointer to the target object.
- **mem_fun_ref(R (T::*f)())** for member functions **R T::f()** that are called through a reference to the target object.
- **mem_fun1(R (T::*f)(A))** for member functions **R T::f(A)** with one argument that are called through a pointer to the target object.
- **mem_fun1_ref(R (T::*f)(A))** for member functions **R T::f(A)** with one argument that are called through a reference to the target object.

The following example draws all graphic objects in a container.

```
class GraphicObject {
public:
  void draw() const;
// ..
}

list<GraphicObject*>  drawing;
for_each(drawing.begin(), drawing.end(), mem_fun(&GraphicObject::draw));
```

Comparing ranges

template <**class** InputIterator1, **class** InputIterator2>
bool *// return true iff [*first1..last1[== [*first2..*(first2+(last1-first1)[; O(n)*
equal(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2);

template <**class** InputIterator1, **class** InputIterator2, **class** BinaryPredicate>
bool *// like above but use comp instead of ==; O(n)*
equal(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, BinaryPredicate comp);

template <**class** InputIterator1, **class** InputIterator2>

```
pair<InputIterator1, InputIterator2> // return (i=first1+k,j=first2+k) where i in [first1..last1[,
// j in [first2..first2+(last1-first2)[ where not(*i==*j); O(n)
mismatch(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2);

template <class InputIterator1, class InputIterator2, class BinaryPredicate>
pair<InputIterator1, InputIterator2> // as above but use comp instead of ==; O(n)
mismatch(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, BinaryPredicate comp);

template <class InputIterator1, class InputIterator2>
bool // return true iff [*first1..*last1[ < [*first2..*last2[ in "dictionary" order; O(n)
lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                        InputIterator2 first2, InputIterator2 last2);
```

Minimum and maximum

```
template <class T>
const T& // return smaller of a and b; O(1)
min(const T& a, const T& b);

template <class T, class Compare>
const T& // return comp(b, a) ? b : a; O(1)
min(const T& a, const T& b, Compare comp);

template <class T>
const T& // return larger of a and b; O(1)
max(const T& a, const T& b);

template <class T, class Compare>
const T& // return comp(a, b) ? b : a;; O(1)
max(const T& a, const T& b, Compare comp);

template <class ForwardIterator>
ForwardIterator      // return i such that *i is minimal in [*first..*last[; O(n)
min_element(ForwardIterator first, ForwardIterator last);

template <class ForwardIterator, class Compare>
ForwardIterator // return i from [first..last[ such that comp(*i,*j) for all j      in [first..last[; O(n)
min_element(ForwardIterator first, ForwardIterator last, Compare comp);

template <class ForwardIterator>
ForwardIterator // return i such that *i is maximal in [*first..*last[; O(n)
max_element(ForwardIterator first, ForwardIterator last);

template <class ForwardIterator, class Compare>
ForwardIterator // return i from [first..last[ such that !(comp(*i,*j)) for j      in [first..last[; O(n)
max_element(ForwardIterator first, ForwardIterator last, Compare comp);
```

7.5.2 Basic Mutating Algorithms

Copying ranges

```
template <class InputIterator, class OutputIterator>
OutputIterator // copy [*first..*last[ to [result..[;
               // does NOT allocate memory; return result+(last-first); O(n)
copy(InputIterator first, InputIterator last, OutputIterator result);

char* // specialization using memmove; O(n)
copy(const char* first, const char* last, char* result);
wchar_t* // specialization using memmove; O(n)
copy(const wchar_t* first, const wchar_t* last, wchar_t* result);

template <class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2 // copy [*first..*last[ to [result-(last-first)..result[
// copying starts with *(result-1) = *(last-1); O(n)
copy_backward(BidirectionalIterator1 first, BidirectionalIterator1 last, BidirectionalIterator2 result);
```

Swapping elements

```
template <class T>
void // exchange values of a and b; O(1)
swap(T& a, T& b);

template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 // exchange contents of [first1..last1[ and [first2..first2+(last1-first1)[; O(n)
swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2);
```

Transforming a range

```
template <class InputIterator, class OutputIterator, class UnaryOperation>
OutputIterator // assign [op(*first)..op(*last)[ to [result..result+(last-first)[;
               // return result+(last-first); O(n)
transform(InputIterator first, InputIterator last, OutputIterator result, UnaryOperation op);

template <class InputIterator1, class InputIterator2, class OutputIterator, class BinaryOperation>
OutputIterator // assign [op(*first1,*first2))..op(*last1,first2+(last1-first1)[
               // to [result..result+(last1-first1)[; return result+(last-first); O(n)
transform(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, OutputIterator result, BinaryOperation op);
```

Replacing elements in a range

```
template <class ForwardIterator, class T>
void // replace every occurrence of old_value in [first..last[ by new_value; O(n)
replace(ForwardIterator first, ForwardIterator last, const T& old_value, const T& new_value);

template <class ForwardIterator, class Predicate, class T>
void // do *i = new_value for each i in [first..last[ where pred(*i); O(n)
replace_if(ForwardIterator first, ForwardIterator last, Predicate pred, const T& new_value);

template <class InputIterator, class OutputIterator, class T>
OutputIterator // copy [*first..last[ to [result..result+(last-first)[, replacing
// each occurrence of old_value by new_value (in [result..[); return result+(last-first); O(n)
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 // as above but replace *i by new_value iff pred(*i); O(n)
replace_copy_if(Iterator first, Iterator last,
                OutputIterator result, Predicate pred, const T& new_value);
```

Filling a range

```
template <class ForwardIterator, class T>
void // *i = value for all i in [first..last[; O(n)
fill(ForwardIterator first, ForwardIterator last, const T& value);

template <class OutputIterator, class Size, class T>
OutputIterator // *i = value for all i in [first..first+n[; return first+n; O(n)
fill_n(OutputIterator first, Size n, const T& value);

template <class ForwardIterator, class Generator>
void // do *i = gen() for all in [first..last[
generate(ForwardIterator first, ForwardIterator last, Generator gen);

template <class OutputIterator, class Size, class Generator>
OutputIterator // *i = gen() for all i in [first..first+n[; return first+n; O(n)
generate_n(OutputIterator first, Size n, Generator gen);
```

Removing elements

The following algorithms do not alter the size of the container corresponding to the input range. Rather, for the non-copying versions, the parameter range $[first \ldots last[$ is rearranged such that a subrange $[first \ldots new\_last[$ contains the result. The iterators in $[new\_last \ldots last[$ are still valid but their content is unspecified.

```
template <class ForwardIterator, class T>
ForwardIterator // rearranges [first..last[ and return new_last such that
```

```
// { *i | i in [first..new_last[ } = { *j | j in [first..last[ && not *j == value }; O(n)
remove(ForwardIterator first, ForwardIterator last, const T& value);

template <class ForwardIterator, class Predicate>
ForwardIterator // as above but use pred(*j) instead of *j==value
remove_if(ForwardIterator first, ForwardIterator last, Predicate pred);

template <class InputIterator, class OutputIterator, class T>
OutputIterator // copy elements *i such that not *i==value from [first..last[ to [result..result+k[
// where k = number of elements in [*first..*last[ that are different from value;
// return result+k; O(n)
remove_copy(InputIterator first, InputIterator last, OutputIterator result, const T& value);

template <class InputIterator, class OutputIterator, class Predicate>
OutputIterator // as above but use pred(*i) instead of *i==value
remove_copy_if(InputIterator first, InputIterator last, OutputIterator result, Predicate pred);

template <class ForwardIterator>
ForwardIterator // remove adjacent duplicates yielding a subrange [first..new_last[
// where *i != *(i+1) for all i; return new_last; O(n)
unique(ForwardIterator first, ForwardIterator last);

template <class ForwardIterator, class BinaryPredicate>
ForwardIterator // as above but use comp(*i,*(i+1)) instead of *i == *(i+1)
unique(ForwardIterator first, ForwardIterator last, BinaryPredicate comp);

template <class InputIterator, class OutputIterator>
OutputIterator // copies elements from [*first..*last[ to [result..new_last[
// such that [result..new_last[ does not contain adjacent duplicates, i.e.
// *i != *(i+1) for all i in [result..new_last-1[; return new_last; O(n)
unique_copy(InputIterator first, InputIterator last, OutputIterator result);

template <class InputIterator, class OutputIterator, class BinaryPredicate>
OutputIterator // as above but use comp(*i,*(i+1)) instead of *i == *(i+1); O(n)
unique_copy(InputIterator first, InputIterator last, OutputIterator result, BinaryPredicate comp);
```

Permuting algorithms

```
template <class BidirectionalIterator>
void // reverse contents of [first..last[; i.e. exchange values of
// first+n and last-(n+1) for all 0<=n<=(last-first)/2; O(n)
reverse(BidirectionalIterator first, BidirectionalIterator last);

template <class BidirectionalIterator, class OutputIterator>
OutputIterator // copy [*first..*last[ to [result..result+(last-first)[ such that
// the result sequence is the reverse of the first sequence; return result+(last-first); O(n)
reverse_copy(BidirectionalIterator first, BidirectionalIterator last, OutputIterator result);

template <class ForwardIterator>
void // rearrange values such that the concatenation [*middle..*last[ [*first..*middle[
// yields the original sequence; e.g. rotate(0,3,4) of 0 1 2 3 4 yields 3 4 0 1 2; O(n)
rotate(ForwardIterator first, ForwardIterator middle, ForwardIterator last);
```

```
template <class ForwardIterator, class OutputIterator>
OutputIterator // as above but rotated range is copied to result, input range is not altered;
// return result+(last-first); O(n)
rotate_copy(ForwardIterator first, ForwardIterator middle, ForwardIterator last, OutputIterator result);

template <class BidirectionalIterator>
bool // transform [first..last[ to the next permutation of [*first..*last[
// in the lexicographical ordering of all permutations of [*first..*last[,
// if it exists (in which case true is returned);
// otherwise, transform [first..last[ into the smallest permutation in the ordering
// and return false; e.g. 0 1 2 3 becomes 0 1 3 2; O(n)
next_permutation(BidirectionalIterator first, BidirectionalIterator last);

template <class BidirectionalIterator, class Compare>
bool // as above but use comp(x,y) instead of operator<(x,y); O(n)
next_permutation(BidirectionalIterator first, BidirectionalIterator last, Compare comp);

template <class BidirectionalIterator>
bool // transform [first..last[ to the previous permutation of [*first..*last[
// in the lexicographical ordering of all permutations of [*first..*last[,
// if it exists (in which case true is returned);
// otherwise, transform [first..last[ into the greatest permutation in the ordering
// and return true; e.g. 0 1 3 2 becomes 0 1 2 3; O(n)
prev_permutation(BidirectionalIterator first, BidirectionalIterator last);

template <class BidirectionalIterator, class Compare>
bool // as above but use comp(x,y) instead of operator<(x,y); O(n)
prev_permutation(BidirectionalIterator first, BidirectionalIterator last, Compare comp);
```

Partitioning ranges

```
template <class BidirectionalIterator, class Predicate>
BidirectionalIterator // reorder elements in [*first..*last[ such that all elements
// in j in [first..middle[ satisfy pred(*j) and the elements in [middle..last[ do not;
// return middle; O(n)
partition(BidirectionalIterator first, BidirectionalIterator last, Predicate pred);

template <class ForwardIterator, class Predicate> // O(n*log(n))
ForwardIterator // as above and preserves relative order in [first..middle[ and [middle..last[;
stable_partition(ForwardIterator first, ForwardIterator last, Predicate pred);
```

Random shuffling and sampling

```
template <class RandomAccessIterator>
void // randomly rearrange [*first..*last[; results are distributed uniformly
// over all permutations; O(n)
random_shuffle(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class RandomNumberGenerator>
```

```
void // as above but use provided random number generator instead of default one; O(n)
// rand must be such that rand(n) returns random number in range [0..n[
random_shuffle(RandomAccessIterator first, RandomAccessIterator last,
               RandomNumberGenerator& rand);

template <class InputIterator, class RandomAccessIterator>
RandomAccessIterator // copy a random sample of size k = min(last-first,out_last-out_first)
// from [*first..*last[ to [out_first..out_first+k[; yields uniformly distributed results;
// return out_first+k; O(n)
random_sample(InputIterator first, InputIterator last,
              RandomAccessIterator out_first, RandomAccessIterator out_last);

template <class InputIterator, class RandomAccessIterator, class RandomNumberGenerator>
RandomAccessIterator // as above but used provided random number generator
// instead of default one; rand must be such thar rand(n) returns a random number
// in the range [0..n[; O(n)
random_sample(InputIterator first, InputIterator last,
              RandomAccessIterator out_first, RandomAccessIterator out_last,
              RandomNumberGenerator& rand);
```

Generalized numeric algorithms

These algorithms become available by including the *numeric* header file.

```
template <class InputIterator, class T>
T // return init + sum of elements in [first..last[; need operator+(const T&,const T&); O(n)
accumulate(InputIterator first, InputIterator last, T init);

template <class InputIterator, class T, class BinaryOperation>
T // as above but use add instead of operator+()
accumulate(InputIterator first, InputIterator last, T init, BinaryOperation add);

template <class InputIterator1, class InputIterator2, class T>
T // return init + sum of (*(first1+i) * *(first2+i)) for all 0<=i<(last1-first1)
inner_product(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, T init);

template <class InputIterator1, class InputIterator2, class T,
          class BinaryOperation1, class BinaryOperation2>
T // as above but use times and add for operator* and operator+, resp.
inner_product(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, T init,
              BinaryOperation1 add, BinaryOperation2 times);

template <class InputIterator, class OutputIterator>
OutputIterator // for each 0<=i<last-first *(result+i) = sum of first i elements from [first..last[;
// result==first is allowed, returns result+(last-first); O(n)
partial_sum(InputIterator first, InputIterator last, OutputIterator result);

template <class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator // as above
partial_sum(InputIterator first, InputIterator last, OutputIterator result, BinaryOperation plus);

template <class InputIterator, class OutputIterator>
OutputIterator // *result = *first and for all 0<i<last-first *(result+i) = *(first+i) - *(first+i-1)
// result==first is allowed; return result+(last-first); O(n)
```

```
adjacent_difference(InputIterator first, InputIterator last, OutputIterator result);

template <class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator // as above      but use subtract instead of operator-
adjacent_difference(InputIterator first, InputIterator last, OutputIterator result,
                    BinaryOperation subtract);
```

7.5.3 Sorting and Searching

The sort algorithms are based on a binary boolean function, by default **`bool operator<(const T&,const T&)`**, that defines a *strictly weak ordering*; i.e. a relation that is irreflexive, antisymmetric and transitive. Moreover, if we define $x \approx y$ iff neither $x < y$ nor $y < x$, then $\approx$ must be an equivalence relation (of elements that are "equivalent" w.r.t. $<$.

A sorting algorithm is called "stable" if the relative order of equivalent elements in the result is the same as in the unsorted input range; **`stable_sort`**, **`merge`** and **`inplace_merge`** are stable but **`sort`** is not.

Sorting ranges

```
template <class RandomAccessIterator>
void // sort range into ascending order; O(n**2); average O(n*log(n))
sort(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void // as above but use comp(x,y) instead of x<y
sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
template <class RandomAccessIterator>

template <class RandomAccessIterator> // usually O(n*log(n))
void // sort range into ascending order; preserving relative position
// of equivalent elements (x equivalent to y iff neither x<y nor y<x)
stable_sort(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void // as above but use comp(x,y) instead of x<y
stable_sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

template <class RandomAccessIterator>
void // rearrange [first..last[ such that [first..middle[ elements are sorted and
// no element in [*middle..*last[ is less than an element in [first..middle[; O(n*log(n))
partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
             RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void // as above but use comp(x,y) instead of x<y
partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
             RandomAccessIterator last, Compare comp);

template <class InputIterator, class RandomAccessIterator>
```

```
RandomAccessIterator // copy smallest N elements from [first..last[
// to [result_first..result_first+N[ where N = min(last-first,result_last-result_first);
// [result_first..result_first+N[ is sorted; return result_first+N; O(n*log(n))
partial_sort_copy(InputIterator first, InputIterator last,
                  RandomAccessIterator result_first, RandomAccessIterator result_last);

template <class InputIterator, class RandomAccessIterator, class Compare>
RandomAccessIterator // as above but use comp(x,y) instead of x<y
partial_sort_copy(InputIterator first, InputIterator last,
                  RandomAccessIterator result_first, RandomAccessIterator result_last,
                  Compare comp);

template <class RandomAccessIterator>
void // rearrange range such that no elements in [first..nth[ are >*nth and
// no elements in [nth..last[ are <*nth; O(n)
nth_element(RandomAccessIterator first, RandomAccessIterator nth, RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void // as above but use comp(x,y) instead of x<y
nth_element(RandomAccessIterator first, RandomAccessIterator nth, RandomAccessIterator last,
            Compare comp);

template <class ForwardIterator>
bool // return true iff [first..last[ is sorted; O(n)
is_sorted(ForwardIterator first, ForwardIterator last);

template <class ForwardIterator, class StrictWeakOrdering>
bool // as above but use comp(x,y) instead of x<y
is_sorted(ForwardIterator first, ForwardIterator last, StrictWeakOrdering comp);
```

Operations on sorted ranges

```
template <class ForwardIterator, class T>
bool // return true if [first..last[ contains element x equivalent with value; O(log(n))
binary_search(ForwardIterator first, ForwardIterator last, const T& value);

template <class ForwardIterator, class T, class Compare>
bool // as above but use comp(x,y) instead of x<y
binary_search(ForwardIterator first, ForwardIterator last, const T& value, Compare comp);

template <class ForwardIterator, class T> // O(log(n))
ForwardIterator // return largest i in [first..last[ such that *j < value for all j in [first..i[
lower_bound(ForwardIterator first, ForwardIterator last, const T& value);

template <class ForwardIterator, class T, class Compare>
ForwardIterator // as above but use comp(x,y) instead of x<y
lower_bound(ForwardIterator first, ForwardIterator last, const T& value, Compare comp);

template <class ForwardIterator, class T>
ForwardIterator // return largest i in [first..last[ such that
// value<*j is false for all j in [first..i[; O(log(n))
upper_bound(ForwardIterator first, ForwardIterator last, const T& value);

template <class ForwardIterator, class T, class Compare>
```

```
ForwardIterator // as above but use comp(x,y) instead of x<y
upper_bound(ForwardIterator first, ForwardIterator last, const T& value, Compare comp);

template <class ForwardIterator, class T>
pair<ForwardIterator, ForwardIterator> // return pair p such that [p.first..p.second[
// is largest subrange containing only elements equivalent to value; more formally
// p.first = lower_bound(first,last,value); p.second = upper_bound(first,last,value); O(log(n))
equal_range(ForwardIterator first, ForwardIterator last, const T& value);

template <class ForwardIterator, class T, class Compare>
pair<ForwardIterator, ForwardIterator> // as above but use comp(x,y) instead of x<y
equal_range(ForwardIterator first, ForwardIterator last, const T& value, Compare comp);

template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator // merge [first1..last1[ and [first2..last2[ in [result..result+N[
// where N = (last1-first1)+(last2-first2); return result+N, O(n)
merge(InputIterator1 first1, InputIterator1 last1,
      InputIterator2 first2, InputIterator2 last2, OutputIterator result);

template <class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
OutputIterator // as above but use comp(x,y) instead of x<y
merge(InputIterator1 first1, InputIterator1 last1,
      InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp);

template <class BidirectionalIterator>
void // merge [first..middle[ and [middle..last[ into [first..last[; O(n)
inplace_merge(BidirectionalIterator first, BidirectionalIterator middle, BidirectionalIterator last);

template <class BidirectionalIterator, class Compare>
void // as above but use comp(x,y) instead of x<y
inplace_merge(BidirectionalIterator first, BidirectionalIterator middle, BidirectionalIterator last,
              Compare comp);
```

Set operations

The following algorithms perform set operations on sorted ranges. Note that all these algorithms apply to **`set<T>`** containers (Section 7.4.5, **`[set<T>::begin()...set<T>::end()[`** is a sorted range) but also to any other sorted range type, e.g. a vector range etc.

A precise description of the behaviour of these algorithms if the order is not total (i.e. if non-identical elements may be equivalent) can be found e.g. in [Aus98].

```
template <class InputIterator1, class InputIterator2>
bool // return true if all elements in [first1..last1[ also occur in [first2..last2[; O(log(n))
includes(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2);

template <class InputIterator1, class InputIterator2, class Compare>
bool // as above but use comp(x,y) instead of x<y
includes(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2,
         Compare comp);
```

```
template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator // [result..result+N[ will contain union of [first1..last1[ and [first2..last2[;
// returns result+N; O(n)
set_union(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2,
          OutputIterator result);

template <class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
OutputIterator // as above but use comp(x,y) instead of x<y
set_union(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2,
          OutputIterator result, Compare comp);

template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator // [result..result+N[ will contain intersection of [first1..last1[ and [first2..last2[;
// returns result+N; O(n)
set_intersection(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2,
                 OutputIterator result);

template <class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
OutputIterator // as above but use comp(x,y) instead of x<y
set_intersection(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2,
                 OutputIterator result, Compare comp);

template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator // [ result..result+N[ will contain elements in [first1..last1[
// but not in [first2..last2[; returns result+N; O(n)
set_difference(InputIterator1 first1, InputIterator1 last1,
                                      InputIterator2 first2, InputIterator2 last2, OutputIterator result);
template <class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
OutputIterator // as above but use comp(x,y) instead of x<y
set_difference(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2,
               OutputIterator result, Compare comp);

template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator // [result..result+N[ will contain elements that are
// either in [first1..last1[ and not in [first2..last2[, or
// in [first2..last2[ and not in [first1..last1[
// returns result+N; O(n)
set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2,
                         OutputIterator result);

template <class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
OutputIterator // as above but use comp(x,y) instead of x<y
set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2,
                         OutputIterator result, Compare comp);
```

Heap operations

A *heap* is a container data structure that provides fast $\mathcal{O}(1)$ access to its largest element. Moreover, inserting and removing an element is also efficient ($\mathcal{O}(\log n)$). Rather than provide a special container type representing heaps, the STL provides a

set of algorithms for manipulating ranges of random access iterators that represent heaps; if $[first \ldots last[$ represents a heap then **`*first`** is its largest element.

Heaps are useful e.g. to represent *priority queues* where elements are inserted in random order but removed in order from largest to smallest[17].

```
template <class RandomAccessIterator>
void // turn [first..last[ into a heap; O(n)
make_heap(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void // as above but use comp(x,y) instead of x<y
make_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

template <class RandomAccessIterator>
void // adds *(last-1) to heap [first..last-1[; making [first..last[ a heap; O(log(n))
push_heap(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void // as above but use comp(x,y) instead of x<y
push_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

template <class RandomAccessIterator>
void // pop the largest elemen (*first) from the heap [first..last[; O(log(n))
// afterwards, [first..last-1[ is still a heap
pop_heap(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void // as above but use comp(x,y) instead of x<y
pop_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

template <class RandomAccessIterator>
void // turn heap [first..last[ into a sorted range; O(n*log(n))
sort_heap(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void // as above but use comp(x,y) instead of x<y
sort_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

template <class RandomAccessIterator>
bool // return true iff [first..last[ represents a heap
is_heap(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class StrictWeakOrdering>
bool // as above but use comp(x,y) instead of x<y
is_heap(RandomAccessIterator first, RandomAccessIterator last, StrictWeakOrdering comp);
```

---

[17] The STL also provides a **`priority_queue`** container adaptor that can be used to implement a heap data type. This adaptor uses the algorithms below.

7.6 Iterator Adaptors

7.6.1 Insert Iterators

Many of the above algorithms blindly copy results to a range that is specified using a single **result** iterator.

Consider e.g. the code for the **copy** algorithm.

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

This only works properly if there is a range $[result \ldots result + (last - first)[$ of valid iterators.

E.g. the following code fragment will result in disaster.

```
int a[] = { 0,1,2,3,4};
vector<int> b;

copy(a, a+5, b);
```

The reason is that $[b \ldots b+5[$ is not a valid range because the vector **b** has no elements.

Of course we could provide an additional **copy_append** algorithm which would append the elements to the target container.

```
template <class InputIterator, class OutputContainer>
OutputContainer& copy_append(InputIterator first, InputIterator last, OutputContainer& result) {
for ( ; first != last; ++first)
  result.push_back(*first);
return result;
}
```

Then we could safely write

```
copy_append(a, a+5, b);
```

However, this solution does not scale well; we would need two versions of many algorithms that make similar assumptions as **copy**.

A better approach, which is adopted in the STL, is to provide an *iterator adaptor* which makes a container **c** look like an output iterator **it** such that ***it = value** is interpreted as **c.push_back(value)**.

```
template <class Container>
class back_insert_iterator: public iterator<output_iterator_tag,void,void,void> {
public:
  explicit back_insert_iterator(Container& x) : container(&x) {}
  back_insert_iterator& operator=(const typename Container::value_type& value) {
```

```
    container->push_back(value);
    return *this;
    }
  back_insert_iterator& operator*() { return *this; } // do nothing
  back_insert_iterator& operator++() { return *this; }
  back_insert_iterator& operator++(int) { return *this; }
protected:
  Container* container;
};

template <class Container>
inline back_insert_iterator<Container> // convenience function
back_inserter(Container& x) {
return back_insert_iterator<Container>(x);
}
```

Note the convenience function. It allows us to write

```
copy(a, a+5, back_inserter(b));
```

which will behave appropriately; elements from the input range will be appended to the container **b**.

The above adaptor is called **back_insert_iterator**. The STL also defines **front_insert_iterator** and **insert_iterator** adaptor classes.

```
template <class Container>
class front_insert_iterator: public iterator<output_iterator_tag,void,void,void> {
public:
  explicit front_insert_iterator(Container& x) : container(&x) {}
  front_insert_iterator<Container>&
  operator=(const typename Container::value_type& value) {
    container->push_front(value);
    return *this;
  }
  front_insert_iterator<Container>& operator*() { return *this; }
  front_insert_iterator<Container>& operator++() { return *this; }
  front_insert_iterator<Container>& operator++(int) { return *this; }
protected:
  Container* container;
};

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

template <class Container>
class insert_iterator: public iterator<output_iterator_tag,void,void,void> {
public:
  insert_iterator(Container& x, typename Container::iterator i): container(&x), iter(i) {}
  insert_iterator<Container>& operator=(const typename Container::value_type& value) {
    iter = container->insert(iter, value);
    ++iter;
    return *this;
    }
```

```
  insert_iterator<Container>& operator*() { return *this; }
  insert_iterator<Container>& operator++() { return *this; }
  insert_iterator<Container>& operator++(int) { return *this; }
protected:
  Container* container;
  typename Container::iterator iter;
};

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

If `it` is a `front_insert_iterator`, `*it = value` will be interpreted as `c.push_front(value)` where `c` is the iterator's underlying container.

Similarly, for an `insert_iterator it`, `*it = value` will result in a call to `c.insert(i,value)` where `c` and `i` are the container, resp. the position in the container, encapsulated by `it`.

7.6.2 Reverse Iterators

Reverse iterators are adaptors for iterators that allow backwards traversal of an iterator range; `++it` on a reverse iterator will execute `--i` where `i` is the iterator underlying the adaptor.

```
template <class Iterator>
class reverse_iterator {
public: // obtain types from underlying iterator
  typedef typename iterator_traits<Iterator>::iterator_category iterator_category;
  typedef typename iterator_traits<Iterator>::value_type value_type;
  typedef typename iterator_traits<Iterator>::difference_type difference_type;
  typedef typename iterator_traits<Iterator>::pointer pointer;
  typedef typename iterator_traits<Iterator>::reference reference;
  typedef Iterator iterator_type;
  typedef reverse_iterator<Iterator> reverse_iterator;

  reverse_iterator() {}
  explicit reverse_iterator(iterator_type x): current(x) {}
  template <class Iter>
  reverse_iterator(const reverse_iterator<Iter>& x) : current(x.current) {}

  iterator_type base() const { return current; }

  reference operator*() const { Iterator tmp = current; return *--tmp; }
  pointer operator->() const { return &(operator*()); }

  reverse_iterator& operator++() { --current; return *this; }
  reverse_iterator  operator++(int) { self tmp = *this; --current; return tmp; }
  reverse_iterator& operator--() { ++current; return *this; }
  reverse_iterator  operator--(int) { self tmp = *this; ++current; return tmp; }
```

```
  reverse_iterator       operator+(difference_type n) const { return self(current − n); }
  reverse_iterator& operator+=(difference_type n) { current −= n; return *this; }
  reverse_iterator       operator−(difference_type n) const { return self(current + n); }
  reverse_iterator& operator−=(difference_type n) { current += n; return *this; }
  reference operator[](difference_type n) const { return *(*this + n); }
protected:
  Iterator current;
};

template <class Iterator>
inline bool operator==(const reverse_iterator<Iterator>& x,
                       const reverse_iterator<Iterator>& y) {
return x.base() == y.base();
}
template <class Iterator>
inline bool operator<(const reverse_iterator<Iterator>& x,
                      const reverse_iterator<Iterator>& y) {
return y.base() < x.base();
}
template <class Iterator>
inline typename reverse_iterator<Iterator>::difference_type
operator−(const reverse_iterator<Iterator>& x, const reverse_iterator<Iterator>& y) {
return y.base() − x.base();
}
template <class Iterator>
inline reverse_iterator<Iterator>
operator+(reverse_iterator<Iterator>::difference_type n, const reverse_iterator<Iterator>& x) {
return reverse_iterator<Iterator>(x.base() − n);
}
```

Thus, the range **`[first..last[`** should contain exactly the same elements as **`[reverse_iterator(first)..reverse_iterator(last)[`** but in the opposite order. Thus e.g. **`*reverse_iterator(first)`** should refer to the same element as **`last-1`** (remember that **`last`** is a "past-the-end" reference). In general, we must have

```
&*reverse_iterator(i) == &*(i−1)
```

and this can be easily verified in the implementation.