Containers

3

Summary: *A container is an object that is used to manage other objects which in this context are called elements of the container. It deals with allocation and deallocation of memory and controls insertion and deletion of elements. The algorithms that work with containers rely on a defined interface of data types and methods which must also be adhered to by user-defined containers if proper functioning of the algorithms is to be guaranteed. The containers* `vector`*,* `list`*, and* `deque` *are described, together with their properties. At the end of the chapter, the peculiarities of cooperation between iterators and containers are discussed.*

In part, the STL containers are typical implicit data types in the sense of Section 1.2. They include `vector`, `list`, and `deque`. Other containers, in contrast, are abstract data types which are implemented by means of the implicit data types. These include `stack`, `queue`, and `priority_queue`.

Further abstract data types are `set`, `map`, `multiset`, and `multimap`. They are implemented by means of so-called red-black trees (Cormen *et al.* (1994)). All abstract data types which do not themselves represent implicit data types can easily be recognized from the fact that they *use* appropriate implicit data types. Abstract data types are described separately in Chapter 4.

Before the individual types of container are introduced, the data types and methods common to all containers will be discussed.

3.1 Data type interface

Each container provides a public set of data types that can be used in a program. The data type `vector<int>::iterator` has already been mentioned on page 9. It can be identical to a pointer type such as `int*`, but this is not compulsory.

The aim of data types is to ensure that the interface to a container in a program is *unique* at compile time. This means that, for example, you can design a several megabytes size vector which is not kept in memory, but is kept as a file on hard disk. Even in this case, you could still use `vector<int>::iterator` as the data type without any danger, but this data type would then be anything but an `int` pointer. The actual implementation of vector element access remains hidden to the user of the container.

Table 3.1 shows the container data types required for user-defined containers and already provided by the containers of the STL. Let `X` be the data type of the container, for example `vector<int>`, and `T` be the data type of a container element, for example `int`. Thus, the type `vector<int>::value_type` is identical to `int`.

Data type	Meaning
`X::value_type`	`T`
`X::reference`	reference to container element
`X::const_reference`	ditto for read-only purposes
`X::iterator`	type of iterator
`X::const_iterator`	ditto, but cannot be used to modify an element
`X::difference_type`	signed integral type (see distance type, page 32)
`X::size_type`	unsigned integral type for size specifications

Table 3.1: Container data types.

3.2 Container methods

Each container provides a public set of methods which can be used in a program. The methods `begin()` and `end()` have already been mentioned and used (pages 5 and 8). Table 3.2 shows the container methods required for user-defined containers and already provided by the STL containers. `X` is the denomination of the container type.

An example of the `swap()` method can be found on page 51. The maximum possible size of a container, determined with `max_size()`, depends among other things on the memory model (only for MS-DOS). A `vector<int>` with a 16-bit `size_t` can contain at most 32 767 elements. The current size, returned by the `size()` function, results from the distance between beginning and end, as calculated by the function `distance(a.begin(), a.end(), n)` described on page 32.

In addition to the above-mentioned methods, there are the relational operators `==`, `!=`, `<`, `>`, `<=`, and `>=`. The first two, `==` and `!=`, are based on comparison of container size and comparison of elements of type `T`, for which `operator==()` must be defined. The remaining four are based on a lexicographic comparison of the elements, for which `operator<()` must be defined as order relation. The relational operators are defined in namespace `std` and make use of the algorithms `equal()` and `lexicographical_compare()` which will be discussed later.

3.2.1 Reversible containers

Reversible containers allow iterators to traverse *backward*. Such iterators may be bidirectional and random access. For these kinds of container, the additional data types

Return type method	Meaning
`X()`	default constructor; creates empty container
`X(const X&)`	copy constructor
`~X()`	destructor; calls the destructors for all elements of the container
`iterator begin()`	beginning of the container
`const_iterator begin()`	beginning of the container
`iterator end()`	position *after* the last element
`const_iterator end()`	ditto
`size_type max_size()`	maximum possible container size (see text)
`size_type size()`	current size of the container (see text)
`bool empty()`	`size() == 0` or `begin() == end()`
`void swap(X&)`	swapping with argument container
`X& operator=(const X&)`	assignment operator
`bool operator==(const X&)`	operator `==`
`bool operator!=(const X&)`	operator `!=`
`bool operator<(const X&)`	operator `<`
`bool operator>(const X&)`	operator `>`
`bool operator<=(const X&)`	operator `<=`
`bool operator>=(const X&)`	operator `>=`

Table 3.2: Container methods.

```
X::reverse_iterator
X::const_reverse_iterator
```

and the methods

```
rbegin()        // points to last element
rend()          // points to fictitious position before the first element
```

are provided which return a reverse iterator.

3.3 Sequences

A sequence is a container whose elements are arranged in a strictly linear way. Table 3.3 shows the methods which must be present for sequences in addition to those of Table 3.2 and which therefore exist in the STL.

Notation for intervals

It is frequently necessary to specify intervals. For this purpose, the usual mathematical interval is used, where square brackets denote intervals including the boundary values, and round parentheses denote intervals excluding the boundary values. Thus, `[i, j)` is an interval including `i` and excluding `j`. In Table 3.3, `X` is the type of a

sequential container; `i` and `j` are of input iterator type; `p` and `q` are dereferenceable iterators; `n` is of type `X::size_type` and `t` is an element of type `X::value_type`.

Return type method	Meaning
`X(n, t)`	Creates a sequence of type `X` with `n` copies of `t`.
`X(i, j)`	Creates a sequence with the elements of the range `[i, j)` copied into the sequence.
`iterator insert(p, t)`	Copies a copy of `t` before the location `p`. The return value points to the inserted copy.
`void insert(p, n, t)`	Copies `n` copies of `t` before the location `p`.
`void insert(p, i, j)`	Copies the elements of the range `[i, j)` before the location `p`. `i, j` refer to another container than that for which `insert()` is called.
`iterator erase(q)`	Deletes the element pointed to by `q`. The returned iterator points to the element immediately following `q` prior to the deletion operation, provided it exists. Otherwise, `end()` is returned.
`iterator erase(q1, q2)`	Deletes the elements of the range `[q1, q2)`. The returned iterator points to the element that pointed to `q2` immediately prior to the deletion operation, provided it exists. Otherwise, `end()` is returned.
`void clear()`	Deletes all elements; corresponds to `erase(begin(), end())`.

Table 3.3: Additional methods for sequences.

The STL contains three kinds of sequential containers, namely `vector`, `list`, and `deque`. A list (`list`) should be used when frequent insertions and deletions are needed somewhere in the middle. A queue with two ends (`deque` = double ended queue) is reasonable when insertion and deletion frequently take place at either end. `vector` corresponds to an array. `deque` and `vector` allow random access to elements.

The above-mentioned operations together with their containers need only constant time. Other operations, however, such as insertion of an element into the middle of a vector or a queue, are more expensive; the average cost increases linearly with the number of already existing elements.

The sequential containers `vector`, `list`, and `deque` provided by the STL offer several other methods, listed later in Table 3.5. The methods take constant time. In addition, there are the operators:

```
template<class T>
bool std::operator==(const Container<T>& x,
                     const Container<T>& y);

template<class T>
bool std::operator<(const Container<T>& x,
                    const Container<T>& y);
```

for comparison, where `Container` can be one of the types `vector`, `list` or `deque`. In addition to the data types of Table 3.1, the types of Table 3.4 are provided.

Data type	Meaning
`X::pointer`	pointer to container element
`X::const_pointer`	ditto, but cannot be used to modify container elements

Table 3.4: Additional data types for `vector`, `list`, *and* `deque`.

3.3.1 Vector

Now that all essential properties of a vector container have been described, let us look at some examples of its application. First, a vector with 10 places is filled with the numbers 0 to 9. At the end, the number 100 is appended, which automatically increases the container size. Subsequently, the vector is displayed in two ways: the first loop uses it as a common array; the second loop uses an iterator.

```
// k3/vector/intvec.cpp
// example for int vector container
#include<vector>
#include<iostream>
using namespace std;

int main() {
    // an int vector of 10 elements
    vector<int> intV(10);

    for(size_t i = 0; i < intV.size(); ++i)
       intV[i] = i;                    // fill vector, random access

    // vector increases on demand
    intV.insert(intV.end(), 100);    // append the number 100

    // use as array
    for(size_t i = 0; i < intV.size(); ++i)
       cout << intV[i] << endl;

    // use with an iterator
    for(vector<int>::iterator I = intV.begin();
                              I != intV.end(); ++I)
       cout << *I << endl;
```

Return type method	Meaning
`void assign(n, t = T())`	Deletes the container elements and subsequently inserts `n` elements `t`.
`void assign(i, j)`	Deletes the container elements and subsequently inserts the elements of the iterator range `[i, j)`.
`reference front()`	Supplies a reference to the first element of a container.
`const_reference front()`	Ditto, but cannot be used to modify container elements.
`reference back()`	Supplies a reference to the last element of a container.
`const_reference back()`	Ditto, but cannot be used to modify container elements.
`void push_back(t)`	Inserts `t` at the end.
`void pop_back()`	Deletes the last element.
`void resize(n, t = T())`	Changes the container size. `n - size()` elements `t` are inserted at the end or `size()-n` elements are deleted at the end, depending on whether `n` is greater or less than the current size.
`reverse_iterator rbegin()`	Returns the begin iterator for backward traversal. This iterator points to the last element.
`const_reverse_iterator rbegin()`	Ditto, but cannot be used to modify container elements.
`reverse_iterator rend()`	Returns the end iterator for backward traversal.
`const_reverse_iterator rend()`	Ditto, but cannot be used to modify container elements.

Table 3.5: Additional methods for `vector`, `list`, *and* `deque`.

```
vector<int> newV(20);              // all elements are 0
cout << " newV = ";

for(size_t i = 0; i < newV.size(); ++i)
   cout << newV[i] << ' ';
```

```
    //swap() from Table 3.2 shows a very fast method for
    // swapping two vectors.
    newV.swap(intV);

    cout << "\n newV after swapping = ";
    for(size_t i = 0; i < newV.size(); ++i)
       cout << newV[i] << ' ';       // old contents of intV

    cout << "\n\n intV         = ";
    for(size_t i = 0; i < intV.size(); ++i)
       cout << intV[i] << ' ';       // old contents of newV
    cout << endl;
}
```

In the next example, the stored elements are of `string` type. In addition, it shows how an element is deleted which leads to a change in the number of elements. All elements following the deleted element shift by one position. This process is a time-consuming operation. Finally, a `reverse_iterator` is used which traverses the container backward.

```
// k3/vector/strvec.cpp
// example for string vector container
#include<vector>
#include<iostream>
#include<string>
using namespace std;

int main() {
    // a string vector of 4 elements
    vector<string> stringVec(4);
    stringVec[0] = "First";
    stringVec[1] = "Second";
    stringVec[2] = "Third";
    stringVec[3] = "Fourth";

    // vector increases size on demand
    stringVec.insert(stringVec.end(), string("Last"));
    cout << "size() = "
         << stringVec.size() << endl;           // 5

    // delete the element 'Second'
    vector<string>::iterator I = stringVec.begin();
    ++I;                                        // 2nd position
    cout << "erase: "
         << *I << endl;
    stringVec.erase(I);     // delete Second
    cout << "size() = "
         << stringVec.size() << endl;           // 4
```

```
      for(I = stringVec.begin(); I != stringVec.end(); ++I)
         cout << *I << endl;

      /* Output:  First
                  Third
                  Fourth
                  Last
      */
      cout << "backwards with reverse_iterator:" << endl;
      for(vector<string>::reverse_iterator
            revI = stringVec.rbegin(); revI != stringVec.rend();
++revI)
             cout << *revI << endl;
  } // main.cpp
```

On average, deletion or insertion of an element at the end of a vector takes constant time, that is $O(1)$ in complexity notation (for example, `pop_back()`). Insertion or deletion of an element somewhere in the middle takes a time proportional to the number of elements that have to be shifted, thus, $O(n)$ for n vector elements.

It should be noted that iterators previously pointing to elements of the vector become invalid when the elements in question are shifted by the insertion or deletion. This also applies when the available space of the vector becomes insufficient for `insert()` and new space is allocated. The reason for this is that after allocation of new, larger memory space all elements are copied into the new space and therefore all old positions are no longer valid.

In addition to the methods of Tables 3.2 to 3.5, `vector` provides the methods of Table 3.6.

3.3.2 List

This example refers to the program on page 41 for the determination of identifiers contained in a file. It makes use of the `Identifier` class described there, with the difference that the identifiers are not written into a file, but into a list which is subsequently displayed:

```
  // k3/list/identify/main.cpp
  #include<iterator>
  #include<fstream>
  #include<list>
  #include"identif.h"

  int main( ) {
      // define and open input file
      std::ifstream Source("main.cpp");

      std::list<Identifier> Identifier_list;

      std::istream_iterator<Identifier> iPos(Source), end;
```

Return type method	Meaning
`reference operator[](n)`	Returns a reference to the `n`th element (usage: `a[n]`, when `a` is the container).
`const_reference operator[](n)`	Ditto, but cannot be used to modify container elements.
`reference at(n)`	Checks if `n` is within the valid range. If yes, a reference to the `n`th element is returned, otherwise an exception is thrown.
`const_reference at(n)`	Ditto, but cannot be used to modify container elements.
`void reserve(n)`	Reserves memory space, so that the available space (capacity) exceeds the currently needed space. Aim: avoiding memory allocation operation during vector use.
`size_type capacity()`	Returns the capacity value (see `reserve()`). `size()` is always less than or equal to `capacity()`.

Table 3.6: Additional vector methods.

```
    if(iPos == end)
       std::cout << "File not found!" << std::endl;

    else
       while(iPos != end)
          // insert identifier and read next one
          Identifier_list.push_back(*iPos++);

    // output
    std::list<Identifier>::const_iterator
                I = Identifier_list.begin();
    while(I != Identifier_list.end())
         std::cout << *I++ << std::endl;
}
```

The structure of the `main()` programs resembles the one on page 43. This resemblance facilitates learning how to use iterators and containers. In contrast to the vector, `insert()` and `erase()` do not invalidate iterators that point to elements of the list, with the exception of an iterator that points to an element to be deleted.

In addition to the methods of Tables 3.2 to 3.5, `list` provides the methods of Table 3.7. Each operation takes constant time ($O(1)$) if not otherwise specified. The predicates mentioned in the table are simply function objects (description on page 21). They determine whether a statement about an element is true or false.

One could, for example, imagine a function object `P` for `Identifier` objects which returns whether the identifier begins with an upper case letter. `remove_if(P)` would then delete all elements of the list that have an upper case initial.

For two of the methods of Table 3.7, namely `merge()` and `splice()`, sample applications are shown.

Merging of sorted lists

Two small sorted lists are to be merged into one big sorted list. After the end of the process, the calling list contains all elements of the two lists, whereas the called list is empty. `merge()` is stable; thus, the relative order of the elements of a list is maintained.

```
// k3/list/merge.cpp
#include<list>
#include<iostream>

// auxiliary function
void displayIntList(const std::list<int> & L) {
    std::list<int>::const_iterator I = L.begin();

    while(I != L.end())
         std::cout << *I++ << ' ';

    std::cout << " size() ="
              << L.size() << std::endl;
}

int main( ) {
    std::list<int> L1, L2;

    // fill lists with sorted numbers
    for(int i = 0; i < 10; ++i) {
        L1.push_back(2*i);        // even numbers
        L2.push_back(2*i+1);      // odd numbers
    }

    displayIntList(L1); // 0 2 4 6 8 10 12 14 16 18 size() =10
    displayIntList(L2); // 1 3 5 7 9 11 13 15 17 19 size() =10

    L1.merge(L2);                 // merge

    displayIntList(L1);
    // 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 size() =20
    displayIntList(L2); // size() =0
}
```

The example first outputs a list of even numbers and a list of odd numbers. After the `merge()` operation, the first list contains all the numbers; the second list is empty.

Return type method	Meaning
`void merge(list&)`	Merges two sorted lists (time complexity $O(n)$).
`void merge(list&, Compare_object)`	Merges two sorted lists, using a `Compare_object` for the comparison of elements ($O(n)$).
`void push_front(const T& t)`	Inserts an element at the beginning.
`void pop_front()`	Deletes the first element.
`void remove(const T& t)`	Removes all elements that are equal to the passed element `t` ($O(n)$).
`void remove_if( Predicate P)`	Removes all elements to which the predicate applies ($O(n)$).
`void reverse()`	Reverses the order of elements in the list ($O(n)$).
`void sort()`	Sorts the elements in the list. Time complexity is $O(n \log n)$. The sorting criterion is the `<` operator defined for the elements.
`void sort( Compare_object)`	as `sort()`, but with the sorting criterion of the Comparison object (see page 22).
`void splice(iterator pos, list& x)`	Inserts the contents of list `x` before `pos`. Afterwards, `x` is empty.
`void splice(iterator p, list&x, iterator i)`	Inserts element `*i` of `x` before `p` and removes `*i` from `x`.
`void splice(iterator pos, list& x, iterator first, iterator last)`	Inserts elements in the range `[first, last)` of `x` before `pos` and removes them from `x`. Calling the same object (that is, `&x == this`), takes constant time, otherwise, the cost is of the order $O(n)$. `pos` must not lie in the range `[first, last)`.
`void unique()`	Deletes identical consecutive elements except for the first one (cost $O(n)$). Application to a sorted list leads to the effect that no element occurs more than once.
`void unique( binaryPredicate)`	Ditto, only that instead of the identity criterion another binary predicate is used.

Table 3.7: Additional methods for lists.

Splicing of lists

The term 'splicing' originates from the nautical cabling technique and denotes the fastening together or uniting of several ropes by tucking several strands of rope or cable into each other. Here, we talk about uniting lists. Of the possibilities listed in Table 3.7, we only look at how to transfer a section of a list into another list. From the previous example, only the line containing the `merge()` operation is substituted with the following program fragment:

```
list<int>::iterator I = L2.begin();
advance(I, 4);                              // 4 steps
L1.splice(L1.begin(), L2, I, L2.end());
```

State of the lists before `splice()`:
L1: *0 2 4 6 8 10 12 14 16 18*
L2: *1 3 5 7 9 11 13 15 17 19*

State of the lists after `splice()`:
L1: *9 11 13 15 17 19 0 2 4 6 8 10 12 14 16 18*
L2: *1 3 5 7*

All elements of list `L2` from position 4 (counting starts with 0) onward up to the end of the list are transferred to the beginning of list `L1`. Afterwards, list `L2` contains only the first four elements, whereas list `L1` has grown by six elements at the beginning.

3.3.3 Deque

Deque is an abbreviation for *double ended queue*. Like a vector, this sequence allows random access iterators and, exactly like a list, it allows insertion and deletion at the beginning or the end in constant time. Insertions and deletions somewhere in the middle, however, are quite costly ($O(n)$), because many elements must be shifted. A deque might be seen as being internally organized as an arrangement of several memory blocks, where memory management is hidden in a similar way to `vector`. During insertion at the beginning or the end, a new block of memory is added whenever available space is no longer sufficient. In addition to the methods of Tables 3.2 to 3.5, `deque` provides the methods of Table 3.8.

3.3.4 showSequence

A remark to start with: `showSequence()` is not an algorithm of the STL, but a sequence display tool written for the examples in this book. The function is defined:

```
// Template for the display of sequences (file include/showseq.h)
#ifndef SHOWSEQ_H
#define SHOWSEQ_H
#include<iostream>
```

Return type method	Meaning
`reference operator[](n)`	Returns a reference to the `n`th element (usage: `a[n]`, when `a` is the container).
`const_reference operator[](n)`	Ditto, but cannot be used to modify container elements.
`reference at(n)`	Returns a reference to the `n`th element, if `n` is within the valid range. Otherwise an exception is thrown.
`const_reference at(n)`	Ditto, but cannot be used to modify container elements.
`void push_front(const T& t)`	Inserts an element at the beginning.
`void pop_front()`	Deletes the first element.

Table 3.8: Additional deque methods.

```
namespace br_stl {
 template<class Container>
 void showSequence(const Container& s, const char* sep = " ",
                   std::ostream& where = std::cout) {
   typename Container::const_iterator iter = s.begin();
   while(iter != s.end())
      where << *iter++ << sep;
   where << std::endl;
 }
}
#endif
```

If nothing different is specified, output is written to `cout`. The sequence is output completely, that is, from `begin()` to (but excluding) `end()`. The `sep` character string separates the individual elements. It defaults to a space if nothing else is specified in the function call. With these definitions, you can simply write

```
br_stl::showSequence(v);
```

in your program to display an `int` vector `v`, instead of

```
std::vector<int>::const_iterator iter = v.begin();
while(iter != v.end()) std::cout << *iter++ << " ";
std::cout << std::endl;
```

The function is neither designed for nor suited to simple C arrays. Its advantage is that because of the shorter notation, programs become more readable. The function template is read into memory with `#include<showseq.h>`. Inclusion of `#include<iostream>` is done by *showseq.h* and is therefore no longer needed in programs using `showSequence()`.

3.4 Iterator categories and containers

In this section, the different iterator categories which are associated to the containers are evaluated, for example in order to select the most effective algorithm possible at compile time. The following example shows how at compile time the correct function for the display of the iterator type is selected from a set of overloaded functions:

```
// k3/iterator/ityp.cpp        determination of the iterator type
#include<string>
#include<fstream>
#include<vector>
#include<iterator>
#include<iostream>

using namespace std;

// template for getting the type (iterator-tag) of an iterator
template<class Iterator>
typename iterator_traits<Iterator>::iterator_category
get_iteratortype(const Iterator&) {
   typename iterator_traits<Iterator>::iterator_category typeobject;
   return typeobject;
}

// overloaded functions
void whichIterator(const input_iterator_tag&) {
    cout << "Input iterator!" << endl;
}

void whichIterator(const output_iterator_tag&) {
    cout << "Output iterator!" << endl;
}

void whichIterator(const forward_iterator_tag&) {
    cout << "Forward iterator!" << endl;
}

void whichIterator(const random_access_iterator_tag&) {
    cout << "Random access iterator!" << endl;
}

// application
int main( ) {
    // In case of basic data types we have to use the iterator_traits template
    int *ip;                          // random access iterator
    // display of iterator type
    whichIterator(get_iteratortype(ip));
    whichIterator(
               iterator_traits<int*>::iterator_category());
```

```
    // define a file object for reading
    // (actual file is not required here)
    ifstream Source;

    // an istream_iterator is an input iterator
    istream_iterator<string> IPos(Source);

    // display of iterator type
    whichIterator(get_iteratortype(IPos));  // or alternatively:
    whichIterator(iterator_traits<istream_iterator<string> >
            ::iterator_category());

   // define a file object for writing
    ofstream Destination;

    // an ostream_iterator is an output iterator
    ostream_iterator<string> OPos(Destination);

    // display of iterator type
    whichIterator(get_iteratortype(OPos));  // or alternatively:
    whichIterator(iterator_traits<ostream_iterator<string> >
            ::iterator_category());

    vector<int> v(10);
    // display of iterator type
    whichIterator(get_iteratortype(v.begin()));
                  // or some other iterator
    whichIterator(iterator_traits<vector<int>::iterator>
            ::iterator_category());
}
```

A further example shows how to write an overloaded function whose selected implementation depends on the iterator type. The task is to output the last n elements of a container by means of the function `showLastElements()`. It is assumed that at least bidirectional iterators can work on the container. Thus, it is sufficient to equip the function with an iterator to the end of the container and the required number.

```
// k3/iterator/iappl.cpp
#include<iostream>
#include<list>
#include<vector>
#include<iterator>

// calling implementation
template<class Iterator>
void showLastElements(Iterator last,
       typename std::iterator_traits<Iterator>::difference_type
n) {
    typename std::iterator_traits<Iterator>::iterator_category
typeobject;
```

```
    showLastElements(last, n, typeobject);
}

/*This function now calls the corresponding overloaded variation, where the selection at
  compile time is carried out by the parameter iterator_category() whose type
  corresponds to an iterator tag. Therefore, the third parameter is an iterator tag object
  constructed by calling its default constructor.
*/

// first overloaded function
template<class Iterator, class Distance>
void showLastElements(Iterator last, Distance n,
                      std::bidirectional_iterator_tag) {
    Iterator temp = last;
    std::advance(temp, -n);

    while(temp != last) {
       std::cout << *temp << ' ';
       ++temp;
    }
    std::cout << std::endl;
}

/*The bidirectional iterator does not allow random access and therefore no iterator arith-
  metic. Only the operators ++ and -- are allowed for moving. Therefore, advance()
  is used to go back n steps and then display the remaining elements. A random access
  iterator allows arithmetic, which makes the implementation of this case slightly easier:
*/

// second overloaded function
template<class Iterator, class Distance>
void showLastElements(Iterator last, Distance n,
                      std::random_access_iterator_tag) {
    Iterator first = last - n;      // arithmetic
    while(first != last)
        std::cout << *first++ << ' ';
    std::cout << std::endl;
}

// main-program
int main( ) {
    std::list<int> L;                       // list
    for(int i=0; i < 10; ++i) L.push_back(i);

    // call of 1st implementation
    showLastElements(L.end(), 5L);    // 5 long

    std::vector<int> v(10);                 // vector
    for(int i = 0; i < 10; ++i) v[i] = i;
```

```
    // call of 2nd implementation
    showLastElements(v.end(), 5);     // 5 int
}
```

This scheme – providing a function as an interface which then calls one of the overloaded functions with the implementation – allows you to use completely different implementations with one and the same function call. This allows you, in a properly designed program, to change a container type without having to modify the rest of the program.

3.4.1 Derivation of value and distance types

The STL is based on the fact that algorithms use iterators to work with containers. However, this also means that inside an algorithm the container and its properties are not known, and that all the required information must be contained in the iterators. The information are determined by means of the iterator traits classes. A short example follows to show how an algorithm is chosen dependent on the iterator type, and how to derive and use value and distance types. Let us assume two different containers, a list and a vector, in which the element order is to be reversed. Only iterators to the beginning and the end of the corresponding containers are passed to the function named `reverseIt()` (to avoid a conflict with `std::reverse()`).

```
// k3/iterator/valdist.cpp
// Determination of value and distance types
#include<showseq.h>
#include<list>
#include<vector>
#include<iterator>

template<class BidirectionalIterator>
void reverseIt(BidirectionalIterator first,
            BidirectionalIterator last) {
     typename std::iterator_traits<BidirectionalIterator>
                                    ::iterator_category typeobject;
    reverseIt(first, last, typeobject);
}

/*Reversing the order means that one element must be intermediately stored. For this, its
  type must be known. Following the well-proven scheme, the function calls the suitable
  implementation for the iterator type:
*/

template<class BidirectionalIterator>
void reverseIt(BidirectionalIterator first,
            BidirectionalIterator last,
            std::bidirectional_iterator_tag) {
    // Use of the difference type to calculate the number of exchanges. The
    // difference type is derived from the iterator type:
```

```
    typename std::iterator_traits<
                   BidirectionalIterator>::difference_type
         n = std::distance(first, last) -1;

    while(n > 0) {
       // The value type is also derived from the iterator type:
       typename std::iterator_traits<BidirectionalIterator>
           ::value_type temp = *first;
       *first++ = *--last;
       *last = temp;
       n -= 2;
    }
}

/*The second implementation uses arithmetic to compute the distance, which much faster,
  but is possible only with random access iterators:
*/

template<class RandomAccessIterator>
void reverseIt(RandomAccessIterator first,
               RandomAccessIterator last,
               std::random_access_iterator_tag) {
    /*Use of the difference type to calculate the number of exchanges. The difference
      type is derived from the iterator type:
    */
    typename std::iterator_traits<RandomAccessIterator>
       ::difference_type n = last -first -1; // arithmetic!

    while(n > 0) {
        // The value type is also derived from the iterator type:
         typename std::iterator_traits<RandomAccessIterator>
            ::value_type temp = *first;
        *first++ = *--last;
        *last = temp;
        n -= 2;
    }
}


/*At first sight, one could think that the algorithm could do without the distance type when
  comparing iterators and stop when first becomes >= last. However, this assumption
  only holds when a > relation is defined for the iterator type at all. For a vector, where
  two pointers point to a continuous memory area, this is no problem. It is, however, im-
  possible for containers of a different kind, such as lists or binary trees.
*/

int main() {
    std::list<int> L;
    for(int i=0; i < 10; ++i) L.push_back(i);
```

```
    reverseIt(L.begin(), L.end());
    br_stl::showSequence(L);

    std::vector<double> V(10);
    for(int i = 0; i < 10; ++i) V[i] = i/10.;
    reverseIt(V.begin(), V.end());
    br_stl::showSequence(V);
}
```

3.4.2 Inheriting iterator properties

When user-defined iterators are built, they should conform to those of the STL. A bidirectional iterator can be written as follows:

```
// user-defined bidirectional iterator using int as value type
class MyIterator
: public std::iterator<
                 std::bidirectional_iterator_tag, int> {
   // program code for operator++(), and so on
}
```

Here `int` may be substituted by a suitable value type, if needed. There can be up to five template parameters: 1. iterator type, 2. value type, 3. distance type, 4. pointer type, 5. reference type. The last three are optional.

3.5 Iterators for insertion into containers

The idiom shown on page 43

```
while(first != last) *result++ = *first++;
```

copies an input range into an output range, where `oPos` and `iPos` in Section 2.2.2 represent output and input iterators for streams. An output stream normally has more than sufficient space for all copied elements. The same idiomatic notation can also be used for the copying of containers; the previous contents of the target container are overwritten:

```
container Source(100), Target(100);
// fill Source with values here

typename container::iterator first  = Source.begin(),
                             last   = Source.end(),
                             result = Target.begin();

// copying of the elements
while(first != last) *result++ = *first++;
```

There can, however, be a problem: this scheme fails when the `Target` container is *smaller* than the `Source` container, because at some time `result` will no longer

be defined. Perhaps the old contents of `Target` should not be overwritten, but should remain intact and the new contents should just be added.

For these purposes, predefined iterators exist which allow insertion. Insert iterators are output iterators.

The insert iterators provide the operators `operator*()` and `operator++()` in both prefix and postfix version, together with `operator=()`. All operators return a reference to the iterator. The first two have no other function. They exist only for keeping the usual notation `*result++ = *last++`:

```
// Implementation of some operators (excerpt)
template <class Container>
class insert_iterator
 : public iterator<output_iterator_tag,
                   typename Container::value_type,
                   typename Container::difference_type> {
  public:
    insert_iterator<Container>& operator*() {return *this;}
    insert_iterator<Container>& operator++() {return *this;}
    insert_iterator<Container>& operator++(int)
    { return *this;}
// ... and so on
};
```

Only the assignment operator calls a member function of the container, which is dependent on the kind of container. Now, let us look at the expression `*result++ = *last++` in detail, remembering that the order of evaluation is from right to left, because unary operators are right-associative. `*last` is the value to be inserted. The call of the first two operators yields a reference to the iterator itself, so that `result` can be substituted successively:

```
result.operator++(int).operator*().operator=(*last++);
       result.operator*().operator=(*last++);
                 result.operator=(*last++);
```

The compiler optimizes the first two calls, so that the task of insertion only remains with the assignment operator. The three different predefined insert iterators described in the next sections differ exactly on this point.

back_insert_iterator

A back insert iterator inserts new elements into a container at the end, making use of the element function `push_back()` of the container, called by the assignment operator:

```
// Implementation of an assignment operator
back_insert_iterator<Container>& operator=(
                  typename Container::const_reference value) {
    // c points to the container (private pointer attribute of the iterator)
```

```
    c->push_back(value);
    return *this;
}
```

The following example shows the application of a back insert iterator in which the numbers 1 and 2 are appended to a vector:

```
// k3/iterator/binsert.cpp
// Insert iterators : back insert
#include<showseq.h>
#include<vector>
#include<iterator>

int main() {
    std::vector<int> aVector(5);            // 5 zeros
    std::cout << "aVector.size() = "
            << aVector.size() << std::endl; // 5
    br_stl::showSequence(aVector);          // 00000

    std::back_insert_iterator<std::vector<int> >
                      aBackInserter(aVector);

    // insertion by means of the operations *, ++, =
    int i = 1;
    while(i < 3)
          *aBackInserter++ = i++;
    std::cout << "aVector.size() = "
              << aVector.size() << std::endl;  // 7

    br_stl::showSequence(aVector);             // 0000012
}
```

The predefined function `back_inserter()` returns a back insert iterator and facilitates passing iterators to functions. Let us assume a function `copyadd()` which copies the contents of one container into another or adds it when the iterator used is an insert iterator:

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

The above program can be integrated with the following lines in which this function is passed the iterator created with `back_inserter()`:

```
    // copying with function back_inserter()
    std::vector<int> aVector2;              // size is 0
```

```
    copyadd(aVector.begin(), aVector.end(),
                             back_inserter(aVector2));
    std::cout << "new: aVector2.size() = "
              << aVector2.size() << std::endl;
    br_stl::showSequence(aVector2);
```

front_insert_iterator

A front insert iterator inserts new elements into a container at the beginning, making use of the member function `push_front()` of the container, called by the assignment operator. Thus, it is very similar to the back insert iterator. In the following example, `list` is used instead of `vector`, because `push_front` is not defined for vectors.

```
// k3/iterator/finsert.cpp
// Insert iterators: front inserter
#include<showseq.h>
#include<list>
#include<iterator>

int main() {
    std::list<int> aList(5);                  // 5 zeros

    std::cout << "aList.size() = "
              << aList.size() << std::endl;   // 5

    br_stl::showSequence(aList);              // 00000

    std::front_insert_iterator<std::list<int> >
                               aFrontInserter(aList);

    // insertion by means of the operations *, ++, =
    int i = 1;
    while(i < 3)
          *aFrontInserter++ = i++;

    std::cout << "aList.size() = "
              << aList.size() << std::endl;   // 7

    br_stl::showSequence(aList);              // 2100000
}
```

The `copyadd()`- example at the end of the section `back_insert_iterator` works in a similar way with the function `std::front_inserter()` (see example *k3/iterator/finserter.cpp*).

insert_iterator

Now, something may have to be inserted not just at the beginning or at the end, but at an arbitrary position in the container. The insert iterator has been designed for this purpose. Since it can also insert at the beginning and at the end, it can also be used

instead of the back and front insert iterators already described. It must be passed the insertion point. For this purpose, the insert iterator uses the member function `insert()` of the container, called by the assignment operator, whose implementation is shown here:

```
// Possible implementation of the assignment operator
insert_iterator<Container>& operator=(
            typename const Container::value_type& value) {
     // iter is a private variable of the insert_iterator object
     iter = theContainer.insert(iter, value);
     ++iter;
     return *this;
}
```

The private variable `theContainer` is a reference to the container, which is passed to the constructor together with the insertion position, as shown in the following example. The insertion position is stored in the private variable `iter`.

```
// k3/iterator/insert.cpp
// Insert iterator
#include<showseq>
#include<vector>
#include<iterator>

int main() {
    std::vector<int> aVector(5);          // 5 zeros

    std::cout << "aVector.size() = "
         << aVector.size() << std::endl;  // 5
    br_stl::showSequence(aVector);        // 0 0 0 0 0

    // insertion by means of the operations *, ++, =
    std::insert_iterator<std::vector<int> >
        aBeginInserter(aVector, aVector.begin());

    int i = 1;
    while(i < 3) *aBeginInserter++ = i++;
    // vector: 1 2 0 0 0 0 0, size() is now 7
    /*In contrast to the front_insert_iterator, the insert-position remains the
      same, i.e. after inserting an element the position is not the beginning of the vector!
    */
    std::insert_iterator<vector<int> >
        aMiddleInserter(aVector, aVector.begin() +
                        aVector.size()/2);

    while(i < 6) *aMiddleInserter++   = i++;
    // vector: 1 2 0 3 4 5 0 0 0 0, size() is now 10

    std::insert_iterator<vector<int> >
```

```
        anEndInserter(aVector, aVector.end());
    while(i < 9) *anEndInserter++    = i++;

    std::cout << "aVector.size() = "
              << aVector.size() << std::endl;  // 13
    br_stl::showSequence(aVector);     // 1 2 0 3 4 5 0 0 0 6 7 8
}
```

tip

Here, the insert iterator is used to insert elements at the beginning, in the middle, and at the end. It should be noted that an insert iterator invalidates references to the container when, for reasons of space, the container is moved to a different memory location! Applied to the above example, this means that the definitions of the insert iterators *cannot* be concentrated at the top shortly after `main()` at *one* point: the `begin()` and `end()` iterators and the size `size()` would be invalid for the second and third iterators immediately after execution of the first one.

The `copyadd()`-example at the end of the section `back_insert_iterator` also works in a similar way with the function `std::inserter(c,p)`. `p` is an iterator into container `c`.