Abstract data types

4

Summary: *Abstract data types and the implicit data types used for their realization have already been generally discussed in Section 1.2. This chapter first deals with the abstract data types* `stack`*,* `queue`*, and* `priority_queue` *which are provided as template classes by the STL. Subsequently, the sorted associative containers* `set`*,* `map`*,* `multiset`*, and* `multimap` *are considered.*

A template class of the kind presented here is also called a *container adaptor* because it adapts an interface. This means that adaptors insert an interface level with changed functionality between the user and the implicit data types. Thus, when you use a stack object, you work via stack methods with the underlying container which can, for example, be a vector.

The container used as an implicit data type is contained as an object in the class of an abstract data type (aggregation). The abstract data type makes use of the methods of the container. This principle is called *delegation*.

4.1 Stack

A stack is a container which allows insertion, retrieving, and deletion only at one end. Objects inserted first are removed last. As an implicit data type, all sequential container classes are allowed which support the operations `back()`, `push_back()`, and `pop_back()`, as shown in the following excerpt:

```
namespace std {
 template <class T,
           class Container = deque<T> >        // default
 class stack {
   public:
     typedef typename Container::value_type value_type;
     typedef typename Container::size_type size_type;
     typedef typename Container container_type;

   protected:
     Container c;
```

```
  public:
    explicit stack(const Container& = Container());

    bool empty()            const  { return c.empty();}
    size_type size()        const  { return c.size(); }
    value_type& top()              { return c.back(); }
    const value_type& top() const  { return c.back(); }
    void push(const value_type& x) { c.push_back(x);  }
    void pop()                     { c.pop_back();    }
};

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

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

There are also the relational operators `!=`, `<=` etc. In particular, you can also choose `vector` or `list` instead of the standard value `deque`. Thus, a `stack<int, vector<int> >` is a stack for `int` values implemented by means of a vector. An example of the application of stacks follows in Section 4.2.

4.2 Queue

A queue allows you to insert objects at one end and to remove them from the opposite end. The objects at both ends of the queue can be read without being removed. Both `list` and `deque` are suitable data types for implementation. The class `queue` provides the following interface:

```
template<class T, class Container = deque<T> >
class queue {
  public:
    explicit queue(const Container& = Container());

    typedef typename Container::value_type value_type;
    typedef typename Container::size_type size_type;
    typedef Container container_type;

    bool empty()              const;
    size_type size()          const;
    value_type& front();             // read value in front
    const value_type& front() const; // read value in front
    value_type& back();                // read value at end
    const value_type& back()  const;  // read value at end
    void push(const value_type& x);   // append x
    void pop();                       // delete first element
  // private/protected parts omitted
};
```

Of course, the underlying implementation is very similar to that of the stack. The operators == and < exist as well. `queue::value_type` and `queue::size_type` are both derived from the type (`deque` or `list`) used for the container. The following short program is intended to show the practical application of queue and stack as simply as possible. More complicated problems will follow later.

```
// k4/div_adt.cpp
#include<stack>
#include<queue>
#include<deque>
#include<list>
#include<vector>
#include<iostream>

int main() {
    std::queue<int, std::list<int> > aQueue;
    int numbers[] = {1, 5, 6, 0, 9, 1, 8, 7, 2};
    const int count = sizeof(numbers)/sizeof(int);

    std::cout << "Put numbers into the queue:" << std::endl;
    for(int i = 0; i < count; ++i) {
         std::cout.width(6); std::cout << numbers[i];
         aQueue.push(numbers[i]);
    }

    std::stack<int> aStack;
    std::cout << "\n\n Read numbers from the queue (same "
                 "order)\n and put them into the stack:"
              << std::endl;

    while(!aQueue.empty()) {
         int Z = aQueue.front();   // read value
         std::cout.width(6); std::cout << Z;
         aQueue.pop();             // delete value
         aStack.push(Z);
    }
    // ... (to be continued)
```

This little program puts a sequence of `int` numbers into a queue, reads them back out, and puts them on a stack. The stack is built with a `deque` (default), whereas the queue uses a list (`list`).

4.3 Priority queue

A priority queue always returns the element with the highest priority. The priority criterion must be specified when creating the queue. In the simplest case, it is the greatest (or smallest) number in the queue. The criterion is characterized by a class of suitable function objects for comparison (see Section 1.6.3).

In a priority queue you could, for example, store pairs consisting of references to print jobs and associated priorities. For simplicity, only `int` elements are used in the example. The continuation of the program of the previous section shows the application, in which the priority queue internally uses a vector and employs the standard comparison type `greater`:

```
// continued from Section 4.2

     std::priority_queue<int, std::vector<int>,
                              std::greater<int> > aPrioQ;
     // greater: small elements first (= high priority)
     // less: large elements first

     std::cout << "\n\n Read numbers from the stack "
                  "(reverse order!)\n"
                  " and put them into the priority queue:"
               << std::endl;

     while(!aStack.empty()) {
          int Z = aStack.top();                    // read value
          std::cout.width(6); std::cout << Z;  // display
          aStack.pop();                            // delete value
          aPrioQ.push(Z);
     }

     std::cout << "\n\n Read numbers from the priority "
                  " queue (sorted order!)" << std::endl;

     while(!aPrioQ.empty()) {
          int Z = aPrioQ.top();                    // read value
          std::cout.width(6); std::cout << Z;  // display
          aPrioQ.pop();                            // delete value
     }
}
```

Because of the priority queue's internal representation as a binary heap for efficiency reasons (see Section 5.7), only implicit data types with random access iterators are suited, for example `deque` and `vector`. `priority_queue` provides the following interfaces, where `Container` and `Compare` denote the data types for the implicit container and the comparison type:

```
template<class T, class Container = vector<T>,
         class Compare = less<Container::value_type> >
class priority_queue {
  public:
    typedef typename Container::value_type value_type;
    typedef typename Container::size_type size_type;
    typedef Container container_type;

    bool empty()            const;
    size_type size()        const;
    const value_type& top() const;
    void push(const value_type& x);
    void pop();
```

The meaning of the above methods corresponds to that of `stack` and `queue`; the constructor, however, looks slightly different:

```
explicit priority_queue(const Compare& x = Compare(),
                        const Container& = Container());
```

The constructor requires a Compare object. If none is passed, an object generated by the default constructor of the Compare class is passed. In the sample program above, this is `greater<int>()`.

```
priority_queue(InputIterator first, InputIterator last,
               const Compare& x = Compare(),
               const Container& = Container());
```

This constructor takes input iterators as the argument, in order to create a priority queue for a large range in one go. This is more efficient than a series of `push()` operations. In our sample program on page 72, a further priority queue could be created by means of the instruction

```
priority_queue<int, vector<int>, greater<int> >
            anOtherPrioQ(numbers, numbers+count);
```

and at the same time be initialized with the whole number array. The name of the array `numbers` is to be taken as a constant pointer, as is usual in C++.

Operators == and < do not exist because the comparison does not seem reasonable and would be expensive in terms of run time behavior. In Section 10.2, a priority queue is used to accelerate sorting processes on sequential files.

4.4 Sorted associative containers

An associative container allows fast access to data by means of a key which need not coincide with the data. For example, the name and address of an employee could be accessed via a personnel number used as a key. In sets and multisets, the data itself is used as a key, whereas in maps and multimaps, key and data are different. The STL provides four types of associative containers:

- `set`: The keys coincide with the data. There are no elements with the same key in the set, that is, a key occurs either once or it does not occur at all.

- `multiset`: The keys coincide with the data. There may be identical keys (elements) in the set, that is, a key can occur not at all, once, or any number of times.

- `map`: The keys do not coincide with the data. For example, the key could be a number (personnel number) by means of which the data (address, salary, ...) can be accessed. Keys can be any kind of objects. In a dictionary, for example, the key could be an English word which is used to determine a foreign language word (the data). `map` maps a set of keys to a set of associated data. The elements of a map container are pairs of keys and data. They describe a binary relation, that is, a relation between elements of two sets. The set of possible keys is called the 'definition range' of the map, the set of associated data is called the 'value range.' The `map` type is characterized by a unique map, because one key is associated with exactly *one* datum. There are no identical keys, that is, a key either does not occur at all or occurs only once.

- `multimap`: A multimap object has the properties described under `map`, with one exception: there may be identical keys. This means that a key can occur not at all, once or any number of times. Unambiguousness is therefore no longer given.

The STL containers store the keys *sorted*, although this is not required by the actual task described in the above points. This is just an implementation detail that allows you to store these containers in a very compact way as balanced binary trees (red-black trees). Because of the sorting, access to the elements is very fast and the tree grows only by the strictly required amount. An alternative, namely hashing, requires an initial assignment of memory, but is even faster in accessing elements (an average of $O(1)$ with sufficient space instead of $O(\log N)$).

This alternative was not incorporated into the STL, since after a certain date all major modifications or extensions were no longer accepted in order not to jeopardize the time scale for standardization of the programming language and its library. Because of their efficiency, hashed associative containers will be described in Chapter 7.

4.4.1 Set

A set is a collection of distinguishable objects with common properties. $\mathbf{N} = \{0, 1, 2, 3, ...\}$, for example, denotes the set of natural numbers. Since the elements are distinguishable, there can be no two identical elements in one set. All sets used in computer programs are finite.

The class `set` supports mapping of sets in the computer. Although the elements of a set in the mathematical sense are not subject to any order, they are nevertheless internally represented in ordered form to facilitate access. The ordering criterion is specified at the creation of a set. If it is not specified, `less<T>` is used by default.

For sets, the STL provides the class template `set`. With regard to the typical operations with sets, such as intersection and union, `set` is subject to several restrictions which, however, are remedied by the extensions described in Chapter 6.

In addition to the data types specified in Table 3.1 and the methods in Table 3.2 and Section 3.2.1, a class `set<Key, Compare>` provides the public interface described in Tables 4.1 to 4.3. Here, `Key` is the type of those elements that also have the function of keys, and `Compare` is the type of the comparison object. In this case, `key_compare` and `value_compare` are identical and are included only for completeness. The difference occurs only later in Section 4.4.3 in the `map` class.

Data type	**Meaning**
`key_type`	`Key`
`value_type`	`Key`
`key_compare`	`Compare`. Standard: `less<Key>`
`value_compare`	`Compare`. Standard: `less<Key>`

Table 4.1: Set data types.

Constructor	**Meaning**
`set()`	Default constructor: creates an empty container, with `Compare()` used as comparison object.
`set(c)`	Constructor: creates an empty container, with `c` used as comparison object.
`set(i, j, c)`	Constructor: creates an empty container, into which subsequently the elements of the iterator range `[i, j)` are inserted by means of the comparison object `c`. The cost is $N \log N$ with N as the number of inserted elements.
`set(i, j)`	As `set(i, j, c)`, but with `Compare()` as comparison object.

Table 4.2: Set constructors.

The right-hand column of Table 4.3 indicates the complexity, where N refers to the number of inserted, deleted, or counted elements. G stands for the current size of the container returned by `size()`. The meaning of some methods can only be fully understood in connection with multisets (see below). For example, `equal_range()`, which for a `set` object `a` is equivalent to the call `make_pair(a.lower_bound(k), a.upper_bound(k))`, supplies only a pair of directly consecutive iterators when applied to a `set` (if k exists).

The `count()` method can yield only 0 or 1. It is included only for compatibility with multisets (`multiset`). All methods that return an iterator or a pair of iterators return constant iterators for constant sets. Methods for constant sets are not specially listed in Table 4.3.

The following example shows the application of a set of type `set`. More complex operations, such as union and intersection will be discussed in Section 5.6 and Chapter 6.

Return type method	**Meaning**	**Complexity**
`key_compare key_comp()`	Returns a copy of the comparison object used for the construction of the `set`.	1
`value_compare value_comp()`	As `key_comp()` (difference only in `map`).	1
`pair<iterator,bool> insert(t)`	Inserts the element `t`, provided that an element with the corresponding key does not yet exist. The `bool` component indicates whether the insertion has taken place; the `iterator` component points to the inserted element or to the element with the same key as `t`.	$\log G$
`iterator insert(p, t)`	As `insert(t)`, with the iterator `p` being a hint as to where the search for inserting should begin. The returned iterator points to the inserted element or the element with the same key as `t`.	$\log G$
`void insert(i,j)`	Inserts the elements of the iterator range `[i, j)`.	$N \log(G + N)$
`size_type erase(k)`	Deletes all elements with a key equal to `k`. The number of deleted elements is returned.	$N + \log G$
`void erase(q)`	Deletes the element pointed to by the iterator `q`.	1
`void erase(p, q)`	Deletes all elements in the iterator range `[p, q)`.	$N + \log G$
`void clear()`	Deletes all elements.	$N + \log G$
`iterator find(k)`	Returns an iterator to an element with the key `k`, provided it exists. Otherwise, `end()` is returned.	$\log G$
`size_type count(k)`	Returns the number of elements with key `k`.	$G + \log G$
`iterator lower_bound(k)`	Returns an iterator to the first element whose key is not less than `k`.	$\log G$
`iterator upper_bound(k)`	Returns an iterator to the first element whose key is greater than `k`.	$\log G$
`pair<iterator, iterator> equal_range(k)`	Returns a pair of iterators between which the keys are equal `k`.	$\log G$

Table 4.3: Set methods.

```
// k4/setm.cpp   Example for sets
#include<set>
#include<showseq.h>

int main() {
   std::set<int> Set;   // comparison object: less<int>()

   for(int i = 0; i < 10; ++i) Set.insert(i);
   for(int i = 0; i < 10; ++i) Set.insert(i); // no effect
   br_stl::showSequence(Set);                 // 0 1 2 3 4 5 6 7 8 9

   /*The display shows that the elements of the set really occur exactly once. In the next
     part of the program, elements are deleted. In the first variation, first the element is
     sought in order to delete it with the found iterator. In the second variation, deletion
     is carried out via the specified key.
   */
   std::cout << "Deletion by iterator\n"
            "Delete which element? (0..9)" ;
   int i;
   std::cin >> i;
   std::set<int>::const_iterator iter = Set.find(i);

   if(iter == Set.end())
      std::cout << i << " not found!\n";
   else {
       std::cout << "The element " << i
                 << " exists" << Set.count(i)      // 1
                 << " times." << std::endl;

       Set.erase(iter);
       std::cout << i << " deleted!\n";
       std::cout << "The element " << i
                 << " exists" << Set.count(i)      // 0
                 << " times." << std::endl;
   }
   br_stl::showSequence(Set);

   /*The count() method yields either 0 or 1. Thus, it is an indicator as to whether an
     element is present in the set.
   */
   std::cout << "Deletion by value\n"
            "Delete which element? (0..9)" ;
   std::cin >> i;
   int Count = Set.erase(i);
   if(Count == 0)
      std::cout << i << " not found!\n";
   br_stl::showSequence(Set);
```

```
    /*A further set NumberSet is not initialized with a loop, but by specifying the range
      to be inserted in the constructor. Suitable iterators for int values are pointers of
      int* type. The name of a C array can be interpreted as a constant pointer to the
      beginning of the array. When the number of array elements is added to this pointer,
      the result is a pointer that points to the position after the last array element. Both
      pointers can be used as iterators for initialization of a set:
    */
    std::cout << "call constructor with iterator range\n";

    // 2 and 1 twice!
    int Array[] = { 1, 2, 2, 3, 4, 9, 13, 1, 0, 5};
    Count = sizeof(Array)/sizeof(Array[0]);

    std::set<int> NumberSet(Array, Array + Count);
    br_stl::showSequence(NumberSet);    // 0 1 2 3 4 5 9 13
}
```

In this example it can also be seen that the occurring elements are displayed only once although duplicates exist in the original array.

4.4.2 Multiset

A multiset behaves like a set with the exception that not just one, but arbitrarily many identical elements may be present. Table 4.4 shows `insert()` as the only method which behaves differently from its counterpart in the set class and has a different return type.

Return type method	**Meaning**	**Complexity**
`iterator insert(t)`	Inserts the element `t` independently of whether an element with the same key already exists. The iterator points to the newly inserted element.	$\log G$

Table 4.4: Multiset: difference from set.

4.4.3 Map

Exactly like a `set`, a `map` is an associative container, in which, however, unlike `set`, keys and associated data are different. Here, the difference between `key_compare` and `value_compare` mentioned on page 75 takes effect. In the declaration of a set container, the types of key and possibly comparison objects must be specified; in `map`, the data type is needed as well:

```
map<int, string, greater<int> > aMap;
```

The definition is a mapping of `int` numbers onto `string` objects, with the numbers internally sorted in descending order. As with `set`, sorting is not a property of the map, but of internal storage. The type of the comparison object can be left out: `map<int, string> aMap` is then the same as `map<int, string, less<int> > aMap`.

The elements of a map container are pairs: the type `value_type` is identical to `key_type` in `set` or `multiset`, whereas `map::value_type` is equivalent to `pair< Key, T>`, with `Key` being the type of key and `T` the type of data.

The `map` class essentially provides constructors with the same parameters and methods with the same names and parameters as the `set` class. The meaning is equivalent; it is sufficient to remember that pairs are stored instead of single values. There are only two exceptions. The method

```
value_compare value_comp();
```

differs in its meaning from the one in `set`. It returns a function object which can be used for comparison of objects of type `value_type` (that is, pairs). This function object compares two pairs on the basis of their keys and the comparison object used for the construction of the `map`. The class `value_compare` is declared inside the class `map`. For example, let us assume two pairs and a map with the following definitions:

```
pair<int, string> p(9921, "algorithms"),
                  q(2726, "data structures");
```

Now, if there is a map `M` which during construction was connected to the comparison object `CK` for the comparison of keys, then the call

```
bool x = M.value_comp()(p,q);
```

is identical to

```
bool x = CK(p.first, q.first);
```

that is, the comparison of the keys stored in `first`. The second exception is the index operator provided in `map`, which also allows you to access the data via the key as an index. The key must not necessarily be a number:

```
// int key
cout << AddressMap[6];              // output of a name

// string key
cout << DictionaryMap["hello"]; // 'Hallo'
```

If during access the key does not yet exist, it is included into the map, inserting an object generated with the default constructor in place of the data! Therefore, before reading with the index operator, check whether the required element exists. *tip* Otherwise, the `map` will inadvertently be filled with objects generated by the default constructor.

In the following example, some names are associated personnel numbers of the `long` type. These numbers are so big that it would not make sense to employ them as an index of an array. After entering a personnel number, the program outputs the corresponding name.

In order to make the program more readable, the data type for mapping names to numbers and the data type for a value pair are renamed by means of `typedef`.

```
// k4/map1.cpp: Example for map
#include<map>
#include<string>
#include<iostream>
using namespace std;

// two typedefs for abbreviations
// comparison object: less<long>()
typedef std::map<long, std::string>  MapType;
typedef MapType::value_type ValuePair;

int main() {
    MapType aMap;

    aMap.insert(ValuePair(836361136, "Andrew"));
    aMap.insert(ValuePair(274635328, "Berni"));
    aMap.insert(ValuePair(260736622, "John"));
    aMap.insert(ValuePair(720002287, "Karen"));
    aMap.insert(ValuePair(138373498, "Thomas"));
    aMap.insert(ValuePair(135353630, "William"));

    // insertion of Xaviera is not executed, because the key already exists.
    aMap.insert(ValuePair(720002287, "Xaviera"));

    /*Owing to the underlying implementation, the output of the names is sorted by
      numbers:
    */
    std::cout << "Output:\n";
    MapType::const_iterator iter = aMap.begin();
    while(iter != aMap.end()) {
       std::cout << (*iter).first << ':'      // number
                 << (*iter).second            // name
                 << std::endl;
          ++iter;
    }

    std::cout << "Output of the name after entering"
                 " the number\n"
              << "Number: ";
    long Number;
    std::cin >> Number;
    iter = aMap.find(Number);              // O(log N), see text
    if(iter != aMap.end())
        std::cout << (*iter).second         // O(1)
                  << ' '
                  << Map[Number]            // O(log N)
                  << std:: endl;
    elsestd:: cout << "Not found!" << std::endl;
}
```

The name is sought by way of the number. This process is of complexity $O(\log N)$, where N is the number of entries. If the entry is found, it can be output directly by dereferencing the iterator.

Another way to access a map element is via the index operator. Here, it can be clearly seen that the index can be an arbitrarily large number which has nothing to do with the number of actual entries – this is completely different from the usual array.

The access `Map[Number]` has the same complexity as `find()`, and we could do without `find()` in the above example if we could be sure that only numbers that actually exist are entered.

If the index operator is called with a non-existing number, it stores this number in the map and uses the default constructor for generating the data (see the exercises). This ensures that the index operator never returns an invalid reference. In our case, an empty string would be entered. In order to prevent this, `find()` is called beforehand.

Exercises

4.1 For a map `m`, data of type `T` and a key `k`, the call `m[k]` is semantically equivalent to

```
(*((m.insert(make_pair(k, T()))).first)).second
```

because an entry is made for a non-existing key. Analyze the expression, in both the case when the key `k` is contained in `m`, and when it is not.

4.2 Is there a difference if `value_type` is written instead of `make_pair` in the previous exercise?

4.4.4 Multimap

`multimap` differs from `map` in the same way as `multiset` differs from `set`: multiple entries of elements with identical keys are possible, for example, the name Xaviera in the sample program of the previous section. Correspondingly, the function `insert(value_type)` does not return a pair `pair<iterator, bool>`, but only an iterator which points to the newly inserted element (compare with `set`/`multiset`).