Various applications 8

Summary: *While the following chapters deal with complex data structures and algorithms, this chapter looks at smaller applications that show how, thanks to the power of the STL, much can be achieved with relatively short programs. The applications are: output of a cross-reference list of identifiers in a text, generation of a permuted index, and search for related concepts of a given term (thesaurus).*

8.1 Cross-reference

The first example is a program for printing a cross-reference list, that is, a list which contains the words or identifiers occurring in the text in alphabetical order, together with the position of their occurrence, in this case, their line numbers. This is the beginning of the cross-reference list that belongs to the following program:

```
_                : 42 51 54
a                : 11 18 20 20 20 22 65 67 71
aKey             : 48 55 61 67
all              : 10
and              : 9
are              : 68 68
avoid            : 9
b                : 18 21 21 21 22
back             : 58
be               : 10 66
because          : 66
begin            : 74
beginning        : 39 69
 ⋮
```

In the simple variation described here, words occurring in comments are output as well. The appropriate data structure is a map container. The value pairs consist of the identifier of type `string` as key and a list of line numbers. Because of the sorted storage, no special sorting process is needed.

```
// k8/crossref.cpp : program for printing cross-references
#include<fstream>
#include<string>
#include<list>
#include<cctype>
#include<showseq.h>
#include<map>

/*To avoid different sorting of upper case and lower case letters, the class Compare is
  used which converts all strings to be compared into lower case, since a corresponding
  function is not provided in the string class:
*/

struct Compare {
   bool operator()(std::string a, std::string b) const {
     for(size_t i=0; i< a.length(); ++i)
        a[i]=std::tolower(a[i]);
     for(size_t i=0; i< b.length(); ++i)
        b[i]=std::tolower(b[i]);
     return a < b;
   }
};

using namespace std;

int main( ) {
    // This program generates its own cross-reference list.
    ifstream Source("crossref.cpp");

    typedef map<string, list<int>, Compare > MAP;
    MAP CrossRef;

    char c;
    int LineNo = 1;

    /*The next section largely corresponds to the operator>>() on page 41. The
      difference lies in the counting of lines.
    */
    while(Source) {
       // find beginning of identifier
       c = '\0';
       while(Source && !(isalpha(c) || '_' == c)) {
             Source.get(c);
             if(c == '\n') ++LineNo;
       }

       string aKey(1,c);

       // collect rest of identifier
       while(Source && (isalnum(c) || '_' == c)) {
```

```
            Source.get(c);
            if(isalnum(c) || '_' == c)
              aKey += c;
         }
         Source.putback(c);     // back to input stream

         if(c)
            CrossRef[aKey].push_back(LineNo);   // entry
      }

      /*Putting the line number in the list utilizes the fact that the
        MAP::operator[]() returns a reference to the entry, even if this has
        still to be created because the key does not yet exist. The entry for the key aKey
        is a list. Since the line numbers are inserted with push_back(), they are in the
        correct order from the very beginning.

        The output of the cross-reference list profits by the sorted storage. The first
        element of a value pair is the identifier (key), the second element is the list
        which is output by means of the known template.
      */

      MAP::const_iterator iter = CrossRef.begin();

      while(iter != CrossRef.end()) {
          cout << (*iter).first;            // identifier
          cout.width(20 - (*iter).first.length());
          cout << ": ";
          br_stl::showSequence((*iter++).second);   // line numbers
      }
}
```

8.2 Permuted index

A permuted index is printed by some journals at the beginning of a new year to facilitate retrieving articles from the previous year using the terms contained in the titles. The permuted index is alphabetically sorted by words in the title and thus facilitates the search for articles on a given subject. Table 8.1 shows an example with the three titles 'Electronic Mail and POP,' 'Objects in the World Wide Web,' and 'Unix or WindowsNT?'

The alphabetical order of the terms in the second column allows quick orientation. Table 8.1 was generated by the following sample program which exploits a map container and its property of sorted storage. A pointer to each relevant word – here, all words beginning with an upper case letter are included – is stored in the map container together with the current title number. Subsequently, the contents may be output only in a formatted way.

	Search term	Page
	Electronic Mail and POP	174
Electronic	Mail and POP	174
	Objects in the World Wide Web	162
Electronic Mail and	POP	174
	Unix or WindowsNT?	12
Objects in the World Wide	Web	162
Objects in the World	Wide Web	162
Unix or	WindowsNT?	12
Objects in the	World Wide Web	162

Table 8.1: Example of a permuted index.

```
// k8/permidx.cpp
// Program for generation of a permuted index
#include<iostream>
#include<vector>
#include<string>
#include<cstring>  // for strcmp()
#include<map>
#include<cctype>

/*The class StringCompare is needed for the creation of a function object for the map
  container.
*/
struct StringCompare {
   bool operator()(const char* a, const char* b) const {
       return std::strcmp(a,b) < 0;
   }
};

using namespace std;

int main() {
    vector<string> Title(3);
    vector<int> Page(Title.size());

    /*Normally, titles and page numbers would be read from a file, but for simplicity, in
      this example both are wired in:
    */
    Title[0] = "Electronic Mail and POP";       Page[0]=174;
    Title[1] = "Objects in the World Wide Web"; Page[1]=162;
    Title[2] = "Unix or WindowsNT?";            Page[2]= 12;

    typedef map<const char*, int, StringCompare> MAP;
    MAP aMap;
```

```
/*All pointers to words that begin with an upper case letter are stored in the map
   container together with the page numbers of the titles. It is assumed that words
   are separated by spaces. An alternative could be to store not the pointers, but the
   words themselves as string objects.

   On average, however, this would require more memory, and a multimap container
   would have to be used, because the same words can occur in different titles. The
   pointers, in contrast, are unique. The same words in different titles have different
   addresses.
*/
for(size_t i = 0; i < Title.size(); ++i) {
    size_t j = 0;

    do {
        const char *Word = Title[i].c_str() + j;
        if(isalpha(*Word) && isupper(*Word))
           aMap[Word] = i;              // entry

        // find next space
        while(j < Title[i].length()
           && Title[i][j] != ' ') ++j;

        // find beginning of word
        while(j < Title[i].length()
           && !isalpha(Title[i][j])) ++j;
    } while(j < Title[i].length());
}

/*The map container is filled, now we need the output. As usual in such cases, the
   formatting requires more program lines than the rest.
*/
MAP::const_iterator I = aMap.begin();
const size_t leftColumnWidth  = 28,
             rightColumnWidth = 30;

while(I != aMap.end()) {
    // determine left column text
    // = first character of title no. (*I).second
    // up to the beginning of the search term
    // which begins at (*I).first.
    const char *begin = Title[(*I).second].c_str();
    const char *end = (*I).first;

    // and output with leading spaces
    cout.width(leftColumnWidth-(end-begin));
    cout << " ";
    while(begin != end)
          cout << *begin++;

    // output right column text
    cout << " ";     // highlight separation left/right
```

```
            cout.width(rightColumnWidth);
            cout.setf(ios::left, ios::adjustfield);   // ranged left
            cout << (*I).first;

            cout.width(4);
            cout.setf(ios::right, ios::adjustfield); // ranged right
            cout << Page[(*I).second]                 // page number
                 << endl;
            ++I;                                     // go to next entry
        }
    }
```

8.3 Thesaurus

A thesaurus is a systematic collection of words and terms that allows you to find terms related to a given concept. The terms can be similar, but they can also represent opposites or antonyms. In this respect, a thesaurus is a counterpart to a dictionary. The dictionary explains the concept belonging to a given term; the thesaurus presents words related by subject and meaning to a given concept.

The thesaurus used in this example was published in its original form by Peter Mark Roget in 1852. It is contained in the file *roget.dat*, which is used by Knuth (1994) for the generation of a directed graph of 1022 nodes and 5075 edges (= references). The file can be obtained via FTP (see page 271).

Instead of building a graph, this section shows how very fast access to related terms is possible with the lower-bound algorithm. A possible application could be in a text processing system, to provide an author with a formulation aid. The lines of the file look as follows:

```
1existence:2 69 125 149 156 166 193 455 506 527
2inexistence:1 4 167 192 194 368 458 526 527 771
3substantiality:4 323 325
4unsubstantiality:3 34 194 360 432 452 458 527
```
and so on.

The numbers after the concept `substantiality` mean that corresponding entries can be found in lines 4, 323, and 325. There are several possibilities to allow fast access. Here, the `lower_bound()` algorithm is employed, which assumes a sorted container and works with the principle of binary search. It finds the first position that can be used for insertion into the container without violating the sorting order. Thus, the algorithm is also suitable for finding an entry in a container.

Three different containers are needed:

- a vector to contain all terms,
- a vector of lists containing the references, and
- a vector that contains the sorting order and is used as an index vector for fast access.

The alternative of not using an index vector and sorting the first two vectors is not chosen, because it is a rather long-winded process to update all references in the lists.

```
// k8/thesaur.cpp : program for the output of terms
// related to a given concept
#include<fstream>
#include<vector>
#include<string>
#include<list>
#include<cctype>
#include<algorithm>
#include<iostream>

struct indirectCompare {
    indirectCompare(const std::vector<std::string>& v) : V(v)
{}

    bool operator()( int x,  int y) const {
        return V[x] < V[y];
    }

    bool operator()( int x, const std::string& a) const {
        return V[x] < a;
    }

    const std::vector<std::string>& V;
};

/*The class indirectCompare compares the corresponding values in the vector V for
  passed indices, and the reference is initialized during construction of the object. The
  second overloaded function operator directly compares a value with a vector element
  whose index is given.
*/

void readRoget(std::vector<std::string>& Words,
               std::vector<std::list<int> >& lists) {
    // see Appendix, pages 265 ff
}

/*The procedure readRoget() reads the file roget.dat and has nothing much to do with
  the STL. It mainly concentrates on analysis and conversion of the data format and has
  therefore been relegated to the Appendix.
*/

using namespace std;

int main( ) {
    const int Maxi = 1022;   // number of entries in roget.dat
```

```
vector<string> Words(Maxi);
vector<list<int> > relatedWords(Maxi);
vector<int> Index(Maxi);

// read thesaurus file
readRoget(Words,relatedWords);

// build index vector
for(size_t i = 0; i < Index.size(); ++i)
   Index[i] = i;

indirectCompare aComparison(Words);     // functor
sort(Index.begin(), Index.end(), aComparison);

/*The index vector now indicates the ordering, so that Words[Index[0]] is the
  first term according to the alphabetical sorting order. This creates the precondition
  for a binary search.
*/

cout << "Search term? ";
string SearchTerm;
getline(cin, SearchTerm);

// binary search
vector<int>::const_iterator TableEnd,
         where = lower_bound(Index.begin(), Index.end(),
                             SearchTerm, aComparison);

/*If the iterator where points to the end of the table, the term was not found.
  Otherwise, a check must be made as to whether the found term matches the
  search term in its first characters. This does not have to be the case, because
  lower_bound() only returns a position which is suitable for sorted insertion.
*/

bool found = true;          // hypothesis to be checked
if(where == TableEnd)
   found = false;
else  {
     // next possible entry is ≥ search term
     // do they match?
    size_t i = 0;
    while(i < Words[*where].length()
         && i < SearchTerm.length()
         && found)
       if(Words[*where][i] != SearchTerm[i])
          found = false;
       else ++i;
}
```

```
    /*If the term is found, the list of references, provided that references exist, is
      'scoured' with the iterator here, and the corresponding terms are displayed on
      screen.
    */

    if(found) {
       cout << "found   : "
            << Words[*where] << endl
            << "related words:\n";

       list<int>::const_iterator
             atEnd = relatedWords[*where].end(),
             here  = relatedWords[*where].begin();

       if(here == atEnd)
          cout << "not found\n";
       else
          while(here != atEnd)
               cout << '\t' << Words[*here++] << endl;
    }
    else  cout << "not found\n";
}
```

To conclude, the output of the program for the search term 'free' is shown:

Search term? free
found : freedom
related words:
cheapness
permission
liberation
subjection
hindrance
facility
will