Appendix A

A.1 Auxiliary programs

A.1.1 Reading the thesaurus file roget.dat

The function `readRoget()` reads the file *roget.dat* according to the given format, in order to build a data structure for a thesaurus (see Section 8.3).

```
void readRoget(std::vector<std::string>& Words,
               std::vector<std::list<int> >& lists) {
    std::ifstream Source("roget.dat");
    assert(Source);              // let's play it safe!

    const int maxbuf = 200;
    char buf[maxbuf];
    char c;
    size_t i;

    while(Source.get(c)) {
       if(c == '*')                  // skip line
          Source.ignore(1000,'\n');
       else
         if(std::isdigit(c)) {
            Source.putback(c);
            Source >> i;                          // current no.
            Source.getline(buf, maxbuf, ':');     // word
            Words[--i] = buf;

            // read line numbers if present,
            // ignoring backslash:
            while(Source.peek() != '\n') {
                int j;
                Source >> j;
```

```
                lists[i].push_front(--j);

                if(Source.peek() == '\\')
                   Source.ignore(1000,'\n');
             }
          }
       }
    }
```

A.1.2 Reading a graph file

The `ReadGraph()` function is used to read a file for constructing a graph according to the format described on page 243. The graph has only an identifier of type `string` for the vertices. The edge parameters can be of any numeric type or of type `Empty` (see page 236).

```
#ifndef GR_INPUT_H
#define GR_INPUT_H
#include<string>
#include<cctype>
#include<graph.h>
#include<fstream>
#include<iostream>
namespace br_stl {

template<class EdgeParamType>
void ReadGraph(Graph<std::string,EdgeParamType>& G,
               const char *Filename) {
    std::ifstream Source;
    Source.open(Filename);
    if (!Source) {    // error check
        std::cerr << "Cannot open "
                  << Filename << "!\n";
        exit(-1);
    }

    while(Source) {
       char c;
       std::string vertex, VertexSuccessor;
       Source.get(c);
       if(isalnum(c)) {
          Source.putback(c);
          Source >> vertex;
          G.insert(vertex);
          // collect successor now, if present
          bool SuccessorExists = false;
```

```
            Source >> c;
            if(c == '<')
                 SuccessorExists = true;
            else
                 Source.putback(c);

            while(SuccessorExists) {
               Source >> VertexSuccessor;
               if(!isalnum(VertexSuccessor[0]))
                  break; // illegal character

               EdgeParamType Par;
               Source >> Par;                  // read parameters
               G.insert(vertex, VertexSuccessor, Par);
            }
         }
         else // skip line
            while(Source && c != '\n') Source.get(c);
      }
   }

} // namespace br_stl
#endif
```

A.1.3 Creation of vertices with random coordinates

The functions of the following sections can be found in the file *gra_util.h*. The prelims are:

```
#ifndef GRAPH_UTILITIES_H
#define GRAPH_UTILITIES_H
#include<place.h>
#include<graph.h>
#include<fstream>
#include<myrandom.h>
#include<string>
#include<iostream>

namespace br_stl {
```

During automatic creation of an undirected graph, a name must be generated for each vertex. The following auxiliary function converts the current number into a string object which is entered as identifier.

```
// auxiliary function for generating strings out of numbers
std::string i2string(unsigned int i) {
    if(i==0) return std::string("0");
```

```
    char buf[] = "0000000000000000";
    char *pos = buf + sizeof(buf) -1; // point to end

    do
      *--pos = i % 10 + '0';
    while(i /=10);
    return std::string(pos);
}
```

The function `create_vertex_set()` creates a number of vertices with random coordinates between 0 and `maxX` or `maxY` in a graph `G` according to its size (`G.size()`).

```
template<class EdgeType>
void create_vertex_set(Graph<Place, EdgeType>& G,
                       int count, int maxX, int maxY) {
    Random xRandom(maxX),
           yRandom(maxY);

    // create vertices with random coordinates
    int i = -1;
    while(++i < count)
      G.insert(Place(xRandom(), yRandom(),i2string(i)));
}
```

A.1.4 Connecting neighboring vertices

This function connects neighboring vertices. Two places i and j are considered neighbors if there is no place located nearer to the mid-point between these two places than the two places themselves.

This definition of neighborhood is certainly arbitrary. It has the advantage that no place remains unconnected. Predefining a maximum distance between two places as a neighborhood criterion has the disadvantage that a point located slightly out of the way might not be connected.

The above definition resembles the definition of neighborhood used in graph theory for triangulation of a graph (Delaunay triangulation, see Knuth (1994)). The Delaunay triangulation postulates that there exists an interval on the mid-perpendicular between two places starting from which any point is nearer to these two places than to any other place. Usually, the mid-point of the two places lies inside this interval, but this is not mandatory.

We will not discuss the Delaunay triangulation algorithm because it is considerably more complicated than the algorithm presented here. Furthermore, we need only to connect neighboring places, not to subdivide the graph into triangles.

```
template<class EdgeType>
void connectNeighbors(Graph<Place, EdgeType>& G) {
    for(size_t i = 0; i < G.size(); ++i) {
       Place iPlace = G[i].first;
```

```
        for(int j = i+1; j < G.size(); ++j) {
           Place jPlace = G[j].first;

           Place MidPoint((iPlace.X()+jPlace.X())/2,
                          (iPlace.Y()+jPlace.Y())/2);

           /*The following loop is not run time optimized. A possible optimization
             could be to sort the places by their x-coordinates so that only a small
             relevant range must be searched. The relevant range results from the fact
             that the places to be compared must lie inside a circle around the mid-
             point whose diameter is equal to the distance between the places i and j.
           */

           size_t k = 0;
           long int e2 = DistSquare(iPlace, MidPoint);

           while(k < G.size()) {      // not run time optimized
              if(k != j && k != i &&
                 DistSquare(G[k].first, MidPoint) < e2)
                     break;
              ++k;
           }

           if(k == G.size()) { // no nearer place found
              EdgeType dist = Distance(iPlace, jPlace);
              G.connectVertices(i, j, dist);
           }
        }
     }
}
```

A.1.5 Creating a LaTeX file

Creation of a figure of a directed graph as a LaTeX file is carried out by the following function. The image size is defined by `xMax` and `yMax`. The scaling factor increases or decreases the scaling of the image.

```
// Only for undirected graphs!
template<class EdgeType>
void createTeXfile(const char * Filename,
                   Graph<Place, EdgeType>& G,
                   double ScalingFactor,
                   int xMax, int yMax) {
    assert(!G.isDirected());
    std::ofstream Output(Filename);

    if(!Output) {
        std::cerr << Filename << " cannot be opened!\n";
```

```
    exit(1);
}

Output  << "%% This is a generated file!\n"
     << "\\unitlength 1.00mm\n"
     << "\\begin{picture}("
     << xMax << ','
     << yMax << ")\n";

for(size_t iv = 0; iv < G.size(); ++iv) {
   // point
   Output << "\\put("
          << G[iv].first.X()*ScalingFactor
          << ','
          << G[iv].first.Y()*ScalingFactor
          << "){\\circle*{1.0}}\n";

   // name of node
   Output << "\\put("
          << (1.0 + G[iv].first.X()*ScalingFactor)
          << ','
          << G[iv].first.Y()*ScalingFactor
          << "){\\makebox(0,0)[lb]{{\\tiny "
          << G[iv].first            // name
          << "}}}\n";

    /*All edges are drawn. To prevent them from appearing twice in the undirected
      graph, they are drawn only in the direction of the greater index.
    */

   typename
   Graph<Place,EdgeType>::Successor::const_iterator
         I = G[iv].second.begin();

   while(I != G[iv].second.end()) {
      size_t n = (*I).first;
      if(n > iv) {              // otherwise ignore
         double x1,x2,y1,y2,dx,dy;
         x1 =  G[iv].first.X()*ScalingFactor;
         y1 =  G[iv].first.Y()*ScalingFactor;
         x2 =  G[n].first.X()*ScalingFactor;
         y2 =  G[n].first.Y()*ScalingFactor;
         dx = x2-x1;
         dy = y2-y1;

         double dist = std::sqrt(dx*dx+dy*dy);
         int wdh = int(5*dist);
         dx = dx/wdh;
         dy = dy/wdh;
```

```
                Output << "\\multiput("
                       << x1 << "," << y1  << ")("
                       << dx << "," << dy  << "){"
                       << wdh
                       << "}{\\circle*{0.1}}\n";
            }
            ++I;
        }
    }
    Output << "\\end{picture}\n";
}
#endif  // GraphUtilities
```

In the sample programs there is also a similar function `createMPfile()` which generates output for MetaPost which then can be converted to PostScript. The print quality is much better.

A.2 Sources and comments

The Silicon Graphics implementation of the STL can be obtained via

`http://www.sgi.com/Technology/STL`

This implementation is not only part of SGI's compiler, but also used in GNU C++. Commercial variations are supplied by several vendors. Besides the source code, the above Internet address also contains the corresponding documentation and other interesting links. The documentation can be freely used, provided that the copyright notice (see `http://www.sgi.com/Technology/STL`) is included. The examples from this book can be found under

`http://www.ubreymann.de/stlbe.html` and
`http://www.informatik.hs-bremen.de/~brey/stlbe.html`

The thesaurus file *roget.dat* and other interesting files and programs dealt with in Knuth (1994) are contained in the *Stanford graphBase*, whose files can be obtained via FTP from `ftp.labrea.stanford.edu`. Under this address, look for directory `sgb`.

A.3 Solutions to selected exercises

This section contains a selection of solutions which should be considered as suggestions. Often, several solutions exist, even though only one (or none) may be indicated.

Chapter 1

1.1 For clearness, the singly-linked list class `slist` is shown in its entirety. `T` is the placeholder for the data type of a list element.

tip Recommendation: After having learned to build class `slist`, put it aside and use only the standard class `list`. It has the same (and more) functions, and it is standardized.

Supplements to `slist` (compared to page 1.4) are:

`erase()`

`clear()`

`empty()`

`size()`

`iterator::operator==()`

`iterator::operator!=()`

copy constructor, destructor, assignment operator.

```
// k1/a4/slist.h : list template for singly-linked lists
// T is a placeholder for the data type of a list element.
#ifndef SIMPLELIST_H
#define SIMPLELIST_H SIMPLELIST_H

#include<cassert>
#include<iterator>
namespace br_stl {

template<class T>
class slist {
  public:
    /*Some types of the class get public names. Then it is possible to use them outside
      the class without knowing the implementation.
    */
    typedef T value_type;
    typedef ptrdiff_t difference_type;
    typedef T* pointer;
    typedef T& reference;
    // see also text

    slist() : firstElement(0), Count(0) {}
    ~slist() { clear();}

    slist(const slist& sl)
    : firstElement(0), Count(0){
      if(!sl.empty()) {
        iterator I = sl.begin();
        push_front(*I++);
        ListElement *last = firstElement;
        while(I != sl.end()) {
          // insert elements at the end to preserve the ordering
```

```
        last->Next = new ListElement(*I++, 0);
        last = last->Next;
        ++Count;
      }
    }
  }

  slist& operator=(const slist& sl) {
    slist temp(sl);
    // swap mgmt.info. swap() see chapter 5
    std::swap(temp.firstElement, firstElement);
    std::swap(temp.Count, Count);
   return *this;
  }

  bool empty() const { return Count == 0;}
  size_t size() const { return Count;}

 /*The implementation of push_front() creates a new list element and inserts it at
    the beginning of the list:
 */
  void push_front(const T& Datum) { // insert at beginning
    firstElement =  new ListElement(Datum, firstElement);
    ++Count;
  }

private:
  struct ListElement {
      T Data;
      ListElement *Next;
      ListElement(const T& Datum, ListElement* p)
      : Data(Datum), Next(p) {}
  };

  ListElement *firstElement;
  size_t Count;

  /*The list consists of list elements whose type is defined inside the list class as a
    nested public class (struct) ListElement. In a structure, direct access to internal
    data is possible, but this is no problem here because the data is located in the
    private section of the slist class. Each list element carries the pertinent data, for
    example a number, together with a pointer to the next list element. firstElement is
    the pointer to the first list element. The class slist provides an iterator type iterator
    which is located in the public section since it is to be publicly accessible. An
    iterator object stores the current container position in the current attribute. The
    methods satisfy the requirements formulated for iterators.
  */
public:
```

```
class iterator {
   friend class slist;
   public:
    typedef std::forward_iterator_tag iterator_category;
    typedef T value_type;
    typedef T* pointer;
    typedef T& reference;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;

     iterator(ListElement* Init = 0)
     : current(Init){}

     T& operator*() {              // dereferencing
         return current->Data;
     }

     const T& operator*() const { // dereferencing
         return current->Data;
     }

     iterator& operator++() {      // prefix
        if(current) // not yet arrived at the end?
           current = current->Next;
        return *this;
     }

     iterator operator++(int) {    // postfix
        iterator temp = *this;
        ++*this;
        return temp;
     }

     bool operator==(const iterator& x) const {
        return current == x.current;
     }

     bool operator!=(const iterator& x) const {
        return current != x.current;
     }

   private:
     ListElement* current; // pointer to current element
}; // iterator

/*Some methods of the slist class use the iterator class:
*/
```

```
    iterator begin() const { return iterator(firstElement);}
    iterator end()   const { return iterator();}

    iterator erase(iterator position) {
       if(!firstElement) return end(); // empty list
       iterator Successor = position;
       ++Successor;

       // look for predecessor
       ListElement *toBeDeleted = position.current,
                   *Predecessor = firstElement;

       if(toBeDeleted != firstElement) {
          while(Predecessor->Next != toBeDeleted)
                Predecessor = Predecessor->Next;
          Predecessor->Next = toBeDeleted->Next;
       }
       else // delete at firstElement
            firstElement = toBeDeleted->Next;
       delete toBeDeleted;
       --Count;
       return Successor;
    }

     void clear() {
         while(begin() != end())
             erase(begin());
     }
};

template<class Iterator>
int operator-(Iterator second, Iterator first) {
  // similar to std::distance(first, second);
  int count = 0;
   /*The difference between the iterators is determined by incrementing first until
     the second iterator is reached. Thus, the condition is that first lies not after the
     second iterator. In other words: second must be reachable by first by means of
     the ++ operator.
   */
  while(first != second
        && first != Iterator()) {
    ++first;
    ++count;
  }
   // In case of inequality, second is not reachable by first
  assert(first == second);
```

```
    return count;
}

} // namespace br_stl
#endif // SIMPLELIST_H
```

Chapter 4

4.1 The best way is to break down the expression step by step, giving temporary objects auxiliary names. The key `k` shall be of type `Key`. First, a pair `P` is created:

```
P = make_pair(k, T());
```

The expression

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

thus becomes

```
(*((m.insert(P)).first)).second
```

Insertion of this pair is carried out only if it does not yet exist. In any case, `insert()` returns a pair `PIB` of type `pair<iterator, bool>`, so that the expression is further simplified to:

```
(*((PIB).first)).second
```

The first element (`first`) is an iterator pointing to the existing, maybe just inserted, element of type `value_type`, that is, `pair<Key,T>`. This iterator will be called `I`:

```
(*I).second
```

Dereferencing this iterator with `operator*()` yields a reference to an object of type `pair<Key,T>`, of which the second (`second`) element of type `T` is now taken.

4.2 No. `value_type` is a `pair`, and the constructor for a `pair` is called.

Chapter 5

5.1

```
template <class InputIterator1, class InputIterator2>
inline bool equal(InputIterator1 first1,
                  InputIterator1 last1,
                  InputIterator2 first2) {
    return mismatch(first1, last1, first2).first == last1;
}
```

5.2
```
template <class InputIterator1, class InputIterator2,
          class BinaryPredicate>
inline bool equal(InputIterator1 first1,
                  InputIterator1 last1,
                  InputIterator2 first2,
                  BinaryPredicate binary_pred) {
    return mismatch(first1, last1, first2,
                    binary_pred).first  == last1;
}
```

5.3
```
template <class ForwardIterator, class Distance>
void rotate_steps(ForwardIterator first,
                  ForwardIterator last,
                  Distance steps)    {// > 0 = right, < 0 = left
    steps %= (last - first);
    if(steps > 0)
        steps = last - first - steps;
    else
        steps = -steps;
    rotate(first, first + steps, last);
}
```

5.4
```
cout << "\n Stability (relative order) violated "
        "for the following value pairs:\n";
vector<double>::iterator stable_Iter1 = stable.begin();
while(stable_Iter1 != stable.end()) {
    // search for counterpart in unstable[]

    vector<double>::iterator unstable_Iter1 =
           find(unstable.begin(), unstable.end(),
                *stable_Iter1);

    if(unstable_Iter1 != unstable.end()) {
      // check all elements following after stable_Iter1 whether they are
      // also found in unstable[] after the position unstable_Iter1
      // (if not: unstable sorting)
       vector<double>::iterator unstable_Iter2,
                                stable_Iter2 = stable_Iter1;

       ++stable_Iter2;
       ++unstable_Iter1;

       while(stable_Iter2 != stable.end()) {
           unstable_Iter2 =
             find(unstable_Iter1, unstable.end(),
                  *stable_Iter2);
```

```
            if(unstable_Iter2 == unstable.end())   // not found?
               cout << (*stable_Iter1)
                    << ' '
                    << (*stable_Iter2)
                    << endl;
            ++stable_Iter2;
         }
      }
      ++stable_Iter1;
   }
```

A.4 Overview of the sample files

The internet sources (see section A.2 on page 271) contain pointers to downloadable files with all the examples in this book. The further directory structure is oriented by the book's chapters, with the names corresponding to the section numbers. Thus, the directory `k1/a3.4` belongs to Chapter 1, Section 3.4. Self-explanatory names, such as `k3/list`, are also often used. The include directory contains the template classes of this book together with auxiliary files for adaptation to the conditions of the compiler used. On the following pages, the sample files are listed together with the page reference for this book.

A.4.1 Files in the include directory

For simplicity, the files needed in many of the examples and therefore in many directories have been transferred into the include directory. This directory should be specified as the first standard include directory.

File	Description	Page
include/checkvec.h	checked vector	196
include/dynpq.h	dynamic priority queue	245
include/graph.h	graphs	236
include/gra_algo.h	algorithms for graphs	254
include/gra_util.h	auxiliary functions for graphs	267
include/gr_input.h	reading of graph files	266
include/hashfun.h	hash address calculation	180
include/hmap.h	hash map	172
include/hset.h	hash set	181
include/iota.h	iota-class	101
include/place.h	class for places	252
include/setalgo.h	set algorithms	161
include/showseq.h	display of sequences	56
include/sparmat.h	sparse matrix	214
include/myrandom.h	class for random numbers	112

Table A.1: Additions to the include directory.

A.4.2 Files for the introductory examples

See Table A.2.

A.4.3 Files for the standard algorithms

The standard algorithms are described in Chapter 5, therefore no table is given. If needed, they can be found in the Contents. All files are located in the directory `k5`.

File	Description	Page
k1/a3.4/mainc.cpp	examples for interplay	6
k1/a3.4/mainstl.cpp	of STL elements	6
k1/a3.4/maint1.cpp		7
k1/a3.4/maint2.cpp		8
k1/a4/slist.h	singly-linked list	10
k1/a4/mainstl2.cpp	example for slist	13
k1/a6/compare.cpp	example for comparison objects	23
k2/identify/identif.h	class for identifiers	41
k2/identify/identif.cpp	implementation of the above	41
k2/identify/main.cpp	application for the above	43
k2/istring.cpp	istream iterator application	37
k3/iterator/binsert.cpp	example for `back_insert_iterator`	65
k3/iterator/binserter.cpp	example for `back_inserter()`	
k3/iterator/finsert.cpp	example for `front_insert_iterator`	66
k3/iterator/finserter.cpp	example for `front_inserter()`	
k3/iterator/insert.cpp	example for `insert_iterator`	67
k3/iterator/inserter.cpp	example for `inserter()`	
k3/iterator/iappl.cpp	selection of implementation dependent on iterator type	59
k3/iterator/ityp.cpp	determination of iterator type	58
k3/iterator/valdist.cpp	determination of value and distance types	61
k3/list/identif.h	see above: k2/identify ...	41
k3/list/identif.cpp	see above: k2/identify ...	41
k3/list/main.cpp	list of identifiers	52
k3/list/merge.cpp	merging of lists	54
k3/list/splice.cpp	splicing of lists	56
k3/vector/intvec.cpp	example with `int` vector	49
k3/vector/strvec.cpp	example with `string` vector	51
k4/div_adt.cpp	abstract data types stack, deque, priority queue	71
k4/map1.cpp	example for a map	80
k4/setm.cpp	example for a set	77

Table A.2: Files for introductory examples (without makefiles and readme-files).

A.4.4 Files for applications and extensions

The files contained in Table A.3 refer to the examples of Chapters 6 to 11. They usually assume the files of Table A.1.

File	Description	Page
k6/mainset.cpp	set algorithms	166
k7/mainseto.cpp	overloaded operators for sets	184
k7/maph.cpp	map with hash map	180
k8/crossref.cpp	cross-reference	186
k8/permidx.cpp	permuted index	188
k8/roget.dat	thesaurus file	190
k8/thesaur.cpp	program for the above	191
k9/a1/strcvec.cpp	string vector with index check	197
k9/a2/matmain.cpp	example with matrix	200
k9/a2/matrix.h	matrix class	198
k9/a2/matrix3d.h	three-dimensional matrix	202
k9/a3/divmat.cpp	various matrix models	209
k9/a3/matrices.h	fixed matrix for different memory models	205
k9/a4/sparse1.cpp	sparse matrix (variation 1)	211
k9/a4/main.cpp	example with sparse matrix	212
k9/a4/mattest.cpp	run-time measurements	220
k9/a4/readme		
k9/a4/stowatch.h	stopwatch class	
k9/a4/stowatch.cpp	implementation of the above	
k10/extsort.cpp	external sorting	223
k10/extsort.h	templates for external sorting	224
k10/extsortq.cpp	external sorting with accelerator	228
k10/extsortq.h	templates for the above	230
k11/analyse/gra1.dat	graph data	243
k11/analyse/gra1u.dat	graph data	
k11/analyse/gra2.dat	graph data	244
k11/analyse/mainint.cpp	graph with integer edge weights	244
k11/analyse/empty.cpp	graph without edge weights	244
k11/dijkstra/gra2.dat	graph data	244
k11/dijkstra/mainplace.cpp	shortest paths (1) in a graph (Figure 11.8)	254
k11/dijkstra/mdi.cpp	shortest paths (2) in a graph	
k11/dynpq/maindpq.cpp	application of the dynamic priority queue	252
k11/toposort/main.cpp	topological sorting	260
k11/toposort/topo.dat	graph data	259

Table A.3: Files for applications and extensions.