Vectors and matrices 9

Summary: *The elements of the STL can easily be used for constructing arrays or vectors in which the access to elements is checked at run time to determine an index overflow. Construction of matrices for different memory models is quite possible, as is shown for C matrices (row-wise storage), FORTRAN matrices (column-wise storage), and symmetric matrices. A class for sparse matrices is implemented by means of an associative container.*

The vectors and matrices of this chapter are for elements of an arbitrary type, i.e. for complex class types as well as basic data types. If vectors and matrices are to be used exclusively for numerical data types like `double` or `complex`, the standard library class `valarray` could be considered (ISO/IEC (1998), Stroustrup (1997)). `valarray` operations are optimized for speed, but (for exactly that reason) do no bounds checks. Sparse matrices as described here cannot be realized with `valarray` in a comparably simple way.

9.1 Checked vectors

The subscript operators of the vector templates of the STL do not carry out an index check. The following example tries to access an invalid vector position:

```
vector<string> stringVec(4);
// ...

stringVec[0] = stringVec[34];   // Error!
```

Obviously this is a nonsensical assignment. If a program goes on working with values generated by erroneous indices, the error is often detected only through consequential errors and is therefore difficult to identify. It is, however, possible to construct a new vector class named, for example, `checkedVector`, that carries out an index check. This class is not part of the STL, but it builds on it.

The principle is straightforward: `checkedVector` *is a* vector which carries out additional checks. In C++, the relation 'is a' is expressed through public inheritance. The derived class must only provide the constructors of the base class and redefine the index operator:

```
// include/checkvec.h : vector class with checked limits
#ifndef CHECKVEC_H
#define CHECKVEC_H
#include<cassert>
#include<vector>

namespace br_stl {

 template<class T>
 class checkedVector : public std::vector<T> {   // inherit
  public:
    // inherited types
    typedef typename checkedVector::size_type size_type;
    typedef typename checkedVector::iterator iterator;
    typedef typename checkedVector::difference_type difference_type;
    typedef typename checkedVector::reference reference;
    typedef typename checkedVector::const_reference const_reference;

    checkedVector() {
    }

    checkedVector(size_type n, const T& value = T())
    : std::vector<T>(n, value) {
    }

    checkedVector(iterator i, iterator j)
    : std::vector<T>(i, j) {
    }

    reference operator[](difference_type index) {
       assert(index >=0
           && index < static_cast<difference_type>(this->size()));
       return std::vector<T>::operator[](index);
    }

   const_reference operator[](difference_type index) const {
       assert(index >=0
           && index < static_cast<difference_type>(this->size()));
       return std::vector<T>::operator[](index);
    }
 };
}
#endif
```

tip

Note: The STL allows inheritance, but does *not* support polymorphism! In this sense, methods of derived classes may be called, but not via pointers or references of the base class type. In the case of vectors, this is certainly no problem, but be aware of it.

`difference_type` is deliberately chosen as the argument type, so that negative erroneous index values are recognized as well. The type `size_type` would lead to an `int` → `unsigned` conversion, and a negative index would be recognized only because it is converted into a significantly large number. Applying this template generates error messages at run time, when the permitted index range is exceeded in either direction. The index check can be switched off with the preprocessor instruction `#define NDEBUG`, if it is inserted before `#include<cassert>`. The following program provokes a run time error by accessing a non-existent vector element:

```
// k9/a1/strvec.cpp
// string vector container with index check
#include<checkvec.h>    // contains checkedVector
#include<iostream>
#include<string>

int main() {
    // a string vector of 4 elements
    br_stl::checkedVector<std::string> stringVec(4);
    stringVec[0] = "first";
    stringVec[1] = "second";
    stringVec[2] = "third";
    stringVec[3] = "fourth";
    std::cout << "provoked program abort:" << std::endl;
    stringVec[4] = "index error";   // error
}
```

Thus, the `checkedVector` class puts a so-called safety wrapper around the vector class. One interface, namely the access to elements of the vector, is adapted to the safety requirements, which is why the `checkedVector` class can be called a kind of ‘vector adaptor.’

9.2 Matrices as nested containers

Besides one-dimensional arrays, two- and three-dimensional matrices are widely used in mathematical applications. Matrices can be build using the `valarray` class of the C++ standard library and related classes. The implementation by means of containers from the STL is also possible, as shown here. Mathematical matrices are special cases of arrays of elements which are of the data types `int`, `float`, `complex`, `rational`, or similar. The `checkedVector` class (Section 9.1) is a one-dimensional matrix in this sense, with the difference that, unlike a normal C array, the class allows safe access via the index operator, as we would also expect for two- and more-dimensional matrix classes. Access to the elements of a one- or more-dimensional matrix object should

- be safe by checking all indices, and
- be carried out via the index operator `[]` (or `[][]`, `[][][]`, ...), so that the usual notation can be maintained.

A possible alternative would be to overload the bracket operator for round parentheses (), which is shown in Section 9.3. It may be argued that it is more pleasing to the eye to write `M(1,17)` instead of `M[1][17]`. When writing new programs, this is really not important. But what if you are responsible for maintaining and servicing existing large programs which use the `[]` syntax? A further argument is that a matrix class should behave as similarly as possible to a conventional C array.

The first requirement is often dismissed, the decrease in efficiency being the justification. This argument is not a hard and fast rule, for more than one reason:

- A correct program is more important than a fast one. As industry practice shows, index errors occur quite frequently. Finding the source of the error is difficult when calculation is continued with erroneous data and the error itself becomes evident only through consequential errors.

- The increased run time caused by checked access is often comparable to further operations relative to the array element, and is sometimes negligible. In the fields of science and engineering, there are programs in which the index check is significantly disadvantageous; however, it depends on the specific case. Only if a program is too slow *because of* the index check, might one consider, after thorough testing, taking the index check out.

9.2.1 Two-dimensional matrices

What is a two-dimensional matrix whose elements are of type `int`? An `int` matrix *is a* vector consisting of `int` vectors! This view allows a significantly more elegant formulation of a matrix class in comparison to the assertion: 'The matrix *has* or consists of mathematical `int` vectors.' The formulation of the *is a* relation as inheritance shows the class `Matrix`. Again, it is not the standard vector container which is employed, but the `checkedVector` class of page 195 derived from it, so that automatic index checking is achieved. Only if no index check is required at all should the `checkedVector` be replaced with the `vector`:

```
// k9/a2/matrix.h
#ifndef MATRIX_H
#define MATRIX_H
#include<checkvec.h>   // checked vector of Section 9.1
#include<iostream>     // for operator<<(), see below

/*matrix as vector of vectors
*/
template<class T>
class Matrix : public br_stl::checkedVector<
                       br_stl::checkedVector<T> > {

   public:
     typedef typename std::vector<T>::size_type size_type;
     Matrix(size_type x = 0, size_type y = 0)
```

```
  : br_stl::checkedVector< br_stl::checkedVector<T> >(x,
      br_stl::checkedVector<T>(y)), rows(x), columns(y) {
  }
```

/*Thus, the `Matrix` class inherits from the `checkedVector` class, with the data type of the vector elements now being described by a `checkedVector<T>` template. With this, the matrix is a nested container that exploits the combination of templates with inheritance.

The constructor initializes the implicit subobject of the base class type (`checkedVector< checkedVector<T> >`) with the correct size `x`.

Exactly as with the standard vector container, the second parameter of the constructor specifies with which value each vector element is to be initialized. Here, the value is no more than a vector of type `checkedVector<T>` and length `y`.

Some simple methods follow for returning the number of rows and columns, initializing all matrix elements with a given value (`init()`), and generation of the identity matrix (`I()`), in which all diagonal elements = 1 and all other elements = 0. For comparison: `init()` does not return anything and `I()` returns a reference to the matrix object, so that the latter method allows chaining of operations:
*/

```
  size_type Rows()    const {return rows; }

  size_type Columns() const {return columns; }

  void init(const T& Value) {
     for (size_type i = 0; i < rows; ++i)
        for (size_type j = 0; j < columns ; ++j)
            operator[](i)[j] = Value; // that is, (*this)[i][j]
  }
```

/*The index operator `operator[]()` is inherited from `checkedVector`. Applied to `i`, it supplies a reference to the `i`th element of the (base class subobject) vector. This element is itself a vector of type `checkedVector<T>`. It is again applied to the index operator, this time with the value `j`, which returns a reference to an object of type `T`, which is then assigned the value.
*/

```
  // create identity matrix
  Matrix<T>& I()     {
     for (size_type i = 0; i < rows; ++i)
        for (size_type j = 0; j < columns ; ++j)
            operator[](i)[j] = (i==j) ? T(1) : T(0);
     return *this;
  }

protected:
  size_type rows,
            columns;
  // here, mathematical operators could follow ...
```

```
};    // class Matrix
#endif
```

Further mathematical operations are omitted, because the point is not to describe a voluminous matrix class, but to show the flexible and varied way in which elements of the STL can be used for the construction of new data structures. In light of this, it is not easy to understand why the C++ standardization committee has chosen a numeric library which is not based on the STL, but is no easier to handle. `Matrix` has no dynamic data outside the base class subobject. Therefore, no special destructor, copy constructor, or an own assignment operator is needed. The corresponding operations of the base class subobject are carried out by the `checkedVector` class or its superclass `vector`.

To facilitate the output of a matrix, we can quickly formulate an output operator which displays a matrix together with its row numbers:

```
template<class T>
inline std::ostream& operator<<(std::ostream& s,
                                const Matrix<T>& m ) {
    typedef typename Matrix<T>::size_type size_type;

    for (size_type i = 0; i < m.Rows(); ++i) {
       s << std::endl << i <<" :  ";
       for (size_type j = 0; j < m.Columns(); ++j)
            s << m[i][j] <<" ";
    }

    s << std::endl;
    return s;
}
#endif      // file matrix.h
```

Further operations and functions can be built following this scheme. Some sample applications show that applying the matrix class is extremely simple (see *k9/a2/matmain.cpp*):

```
Matrix<float> a(3,4);
a.init(1.0);              // set all elements = 1
cout << " Matrix a:\n" << a;
```

The output of this simple program part is

```
Matrix a:
0 : 1 1 1 1
1 : 1 1 1 1
2 : 1 1 1 1
```

Chaining of operations by returning the reference to the object is shown in the line

```
cout << "\n Identity matrix:\n" << a.I();
```

where `a.I()` returns the matrix object so that `template<class T> ostream& operator<<(ostream& s, const Matrix<T>& m)` can be called. As with a simple C array, the index operator can be chained, but with the advantage that the index is checked for limits:

```
Matrix<float> b(4,5);
for(int i=0; i< b.Rows(); ++i)
    for(int j=0; j< b.Columns(); ++j)
         b[i][j] = 1+i+(j+1)/10.;      // index operator
    cout << "\n Matrix b:\n" << b;
```

Output:

```
Matrix b:
0 : 1.1 1.2 1.3 1.4 1.5
1 : 2.1 2.2 2.3 2.4 2.5
2 : 3.1 3.2 3.3 3.4 3.5
3 : 4.1 4.2 4.3 4.4 4.5
```

Owing to the check in `operator[]()`, an assignment of the kind `b[100][99] = 1.0` leads to the erroneous program being aborted. Now, how do element access and index check work? Let us consider the following example:

```
b[3][2] = 1.0;
```

Access is very simple; both indices are checked. Explaining how it works, however, is not that easy. In order to see what happens, we now rewrite `b[3][2]` and resolve the function calls:

```
(b.checkedVector<checkedVector<float> >
                              ::operator[](3)).operator[](2)
```

The anonymous base class subobject is a `checkedVector` whose `[]` operator is called with the argument '3.' The elements of the vector are of type `checkedVector <float>`; that is, a reference to the third `checkedVector<float>` of the base class subobject is returned. If, for simplicity, we call the return value `X`, then

```
X.operator[](2)
```

is executed, which means no more than executing the index operation `operator[]()` for a `checkedVector<float>` with the result `float&`, that is, a reference to the sought element. In each of these index operator calls, the limits are checked in a uniform way. Apart from the equivalent definition for constant objects, there exists *only one* definition of the index operator! ***tip***

9.2.2 Three-dimensional matrix

The scheme used for two-dimensional matrices can now easily be extended for matrices of arbitrary dimensions. Here, as a conclusion, an example for three dimensions is given. What is a three-dimensional matrix whose elements are of type `int`?

The question can easily be answered in analogy to the previous section. A three-dimensional `int` matrix *is a* `vector` of two-dimensional `int` matrices! The formulation of the *is a* relation as inheritance is shown by the `Matrix3D` class:

```
// k9/a2/matrix3d.h      3D matrix as vector of 2D matrices
#ifndef MATRIX3D_H
#define MATRIX3D_H
#include"matrix.h"

template<class T>
class Matrix3D : public br_stl::checkedVector<Matrix<T> > {
  public:
    typedef typename std::vector<T>::size_type size_type;
    Matrix3D(size_type x = 0, size_type y = 0,
             size_type z = 0)
    : br_stl::checkedVector<Matrix<T> >(x, Matrix<T>(y,z)),
      rows(x), columns(y), zDim(z) {
    }

    /*The constructor initializes the base class subobject, a checkedVector of length
      x, whose elements are matrices. Each element of this vector is initialized with a
      (y,z) matrix.
    */
    size_type Rows()    const { return rows;}
    size_type Columns() const { return columns;}
    size_type zDIM()    const { return zDim;}

    /*The other methods resemble those of the Matrix class. The init() method
      needs only one loop over the outermost dimension of the three-dimensional
      matrix, because operator[](i) is of type &Matrix<T> and therefore
      Matrix::init() is called for each two-dimensional submatrix:
    */
    void init(const T& Value) {
       for (size_type i = 0; i < rows; ++i)
           operator[](i).init(Value);
    }
  protected:
    size_type rows,
              columns,
              zDim;          // 3rd dimension
    // here, mathematical operators could follow ...
};
#endif
```

Since, like `Matrix`, `Matrix3D` has no dynamic data outside the base class subobject, no special destructor, copy constructor, or own assignment operator is needed. The corresponding operations for the base class subobject are carried out by the `checkedVector` class itself. The index operator is inherited. Three-dimensional matrices can be defined and used in a simple way, for example:

```
// Excerpt from k9/a2/matmain.cpp
#include"matrix3d.h"

int main() {
    Matrix3D<float> M3(2,4,5);

    for (i=0; i< M3.Rows(); ++i)
        for (int j=0; j< M3.Columns(); ++j)
             for (int k=0; k< M3.zDIM(); k++)
              // chained index operator on the left hand side
              M3[i][j][k] = 10*(i+1)+(j+1)+(k+1)/10.;

    std::cout << "\n 3D matrix:\n";
    for (i=0; i< M3.Rows(); ++i)
       std::cout << "Submatrix " << i
                 << ":\n"
                 << M3[i];
    // ... and so on
```

Since for `M3[i]`, as with a two-dimensional matrix, the output operator is already defined, the output only needs one loop level. The result is:

```
3D matrix:
Submatrix 0:
0 : 11.1 11.2 11.3 11.4 11.5
1 : 12.1 12.2 12.3 12.4 12.5
2 : 13.1 13.2 13.3 13.4 13.5
3 : 14.1 14.2 14.3 14.4 14.5

Submatrix 1:
0 : 21.1 21.2 21.3 21.4 21.5
1 : 22.1 22.2 22.3 22.4 22.5
2 : 23.1 23.2 23.3 23.4 23.5
3 : 24.1 24.2 24.3 24.4 24.5
```

An index error can easily be provoked and is 'rewarded' with program abortion, no matter in which of the three dimensions the error occurs. The functioning of the index operator can be described analogous to the `Matrix` class, but there is one more chained operator call. Let us, for example, reformulate an access `M[1][2][3]`:

```
M.checkedVector<Matrix<float> >::
         operator[](1).operator[](2).operator[](3)
```

The first operator returns something of type `Matrix<float>&` or, more precisely, a reference to the first element of the `checkedVector` subobject of `M`. For readability, we now abbreviate the returned 'something' with `Z` and obtain

```
Z.operator[](2).operator[](3)
```

We know that a reference is only another name (an alias), so that, in the end, `Z` represents a matrix of type `Matrix<float>`. We have already seen that a `Matrix<float>` is a vector of type `checkedVector<checkedVector<float> >`, from which `operator[]()` was inherited. This operator is now called with the argument '2' and returns a result of type `checkedVector<float>&` which, for brevity, will be called '`X`':

```
X.operator[](3)
```

The rest is easy when we think back to the end of Section 9.2.1. Here too, as with the `Matrix` class, access to an element is simpler than the underlying structure.

9.2.3 Generalization

The method for construction of classes for multi-dimensional matrices can easily be generalized: an n-dimensional matrix can always be considered as a vector of $(n-1)$-dimensional matrices, the existence of a class for $(n-1)$-dimensional matrices is assumed. In practice, however, four- and higher-dimensional matrices are seldom employed. The index operator, assignment operator, copy constructor, and destructor do not have to be written, they are provided by the `vector` class; whereas the constructor, the initialization methods, and the required mathematical operators still have to be written.

9.3 Matrices for different memory models

This section will show how matrices for different memory layouts can easily be implemented by means of the STL programming methodology. Here, for a change, the index operator is realized with round parentheses, that is, by overlaying the function operator `operator()()`, because otherwise, an auxiliary class would be needed.

Different memory models can play a role when matrices from or in FORTRAN programs are to be processed, for example when FORTRAN matrix subroutines are called from within a C++ program. The matrices of the previous section are vectors which, depending on the allocator, are not necessarily stored in memory one after the other. Each matrix of this section is, however, mapped to a linear address space, the reason for which a vector container is chosen as a basis. This address space shall be of fixed, unchangeable size, which is expressed by the name `fixMatrix` for the matrix class.

The position of a matrix element `X[i][j]` inside the vector container depends, however, on the kind of storage. Three cases will be discussed:

- C memory layout
  Storage occurs *row-wise*, that is, row 0 lies at the beginning of the container. It

is followed by row 1, and so on. The linear order of the nine elements M_{ij} of a matrix M with three rows and three columns is as follows:

$$M_{00}, M_{01}, M_{02}, M_{10}, M_{11}, M_{12}, M_{20}, M_{21}, M_{22}$$

- FORTRAN memory layout
  In the FORTRAN programming language, storage occurs *column-wise*. Column 0 lies at the beginning of the container, followed by column 1, and so on. The linear order of the nine elements of a matrix with three rows and three columns is therefore:

  $$M_{00}, M_{10}, M_{20}, M_{01}, M_{11}, M_{21}, M_{02}, M_{12}, M_{22}$$

- Memory layout for symmetric matrices
  A symmetric matrix M satisfies the condition $M = M^{\mathrm{T}}$. The raised T stands for 'transposed matrix' and means that $M_{ij} = M_{ji}$ holds for all elements. It follows that a symmetric matrix is quadratic, that is, it has as many rows as columns. Furthermore, it follows that by exploiting the symmetry, one needs only slightly more than half the memory, compared with an arbitrary square matrix. For example, for a symmetric matrix with three rows and three columns, it is sufficient to store the following six instead of nine elements:

  $$M_{00}, M_{01}, M_{11}, M_{02}, M_{12}, M_{22}$$

  An element M_{10} must be searched for at position 1 of the container, where the associated element M_{01} is located.

To implement all three possibilities in a flexible way using the STL, a class `fixMatrix` is defined which provides the most important methods of a matrix, namely the constructor and methods for determining number of rows and columns, together with an operator for accessing individual elements, implemented here by means of the overloaded function operator:

```
// excerpt from k9/a3/matrices.h
template<class MatrixType>
class fixMatrix {
  public:
    typedef typename MatrixType::ValueType ValueType;
    typedef typename MatrixType::IndexType IndexType;
    typedef typename MatrixType::ContainerType ContainerType;

    fixMatrix(IndexType z, IndexType s)
    : theMatrix(z,s,C), C(theMatrix.howmany()) {
    }

    IndexType Rows() const { return theMatrix.Rows();}

    IndexType Columns() const { return theMatrix.Columns();}
```

```
      ValueType& operator()(IndexType z, IndexType s) {
         return theMatrix.where(z,s);
      }
      // ... further methods and operators

    private:
      MatrixType theMatrix;          // determines memory layout
      ContainerType C;               // container C
};
```

The kind of data storage is undefined; it is determined by the placeholder `MatrixType` which is supposed to supply the required properties. The requirements for `MatrixType` result from `fixMatrix`:

- Data types must be provided for the container, the elements to be stored, and the data type of the index.
- The method `howmany()` is used to determine the size of the container.
- The method `where()`, when applied to the object which determines the matrix type, returns a reference to the sought element.
- `Rows()` and `Columns()` methods return the corresponding number.

What is still needed is a proper formulation of the matrix types for the above-mentioned possibilities of element order. Properties common to all three types are formulated as a superclass which is parametrized with the value and index types. In this superclass, the container type is defined as `vector`.

```
#include<cassert>              // used in subclasses
#include<vector>

template<class ValueType, class IndexType>
class MatrixSuperClass {
  public:
    // public type definitions
    typedef ValueType ValueType;
    typedef IndexType IndexType;
    // define vector as container type:
    typedef vector<ValueType> ContainerType;

    IndexType Rows()  const { return rows;}

    IndexType Columns() const { return columns;}

  protected:
    MatrixSuperClass(IndexType z, IndexType s,
                     ContainerType& Cont)
    : rows(z), columns(s), C(Cont) {
    }
```

```
      ContainerType &C;

    private:
      IndexType rows, columns;
  };
```

Because of the `protected` constructor, `MatrixSuperClass` is an abstract class. Outside the derived class, no single object of type `MatrixSuperClass` can be instantiated. With the same result, one could have declared the functions `howmany()` and `where()` common to all as purely virtual methods. The resulting advantage of a compulsory definition of an interface for all derived classes would, however, be overcome by the cost of an internal management table for virtual functions. This is the reason why this alternative is not implemented. Furthermore, it is neither usual nor necessary to access matrices via superclass pointers or references. See also the hint on page 196.

The reference to the container which is physically located in the `fixMatrix` class allows derived classes to access it. The following sections present the outstanding peculiarities.

9.3.1 C memory layout

In the following, `r` stands for 'row' and `c` for 'column.' `CMatrix` inherits, as described, from `MatrixSuperClass`.

```
  template<class ValueType, class IndexType>
  class CMatrix : public MatrixSuperClass<ValueType,IndexType>
{
    public:
     CMatrix(IndexType r, IndexType c,
           typename CMatrix::ContainerType& C)  // inherited type
     : MatrixSuperClass<ValueType,IndexType>(r,c,C) {
     }

     // The size of the vector can easily be calculated:
     IndexType howmany() const {
         return this->Rows()*this->Columns();
     }

     /*The position of an element with the indices r and c is calculated in the where()
       method. Checking of index limits in the vector container is only possible to a limited
       extent, because the check could only be carried out against the entire length (Rows
       × Columns). Therefore, a checkedVector is not sufficient, and the index check
       is carried out directly inside the where() method.
     */

     ValueType& where(IndexType r, IndexType c) const {
        assert(r < this->Rows() && c < this->Columns());
        return this->C[r * this->Columns() + c];
    }
};  // CMatrix
```

A simple program shows the application in which the `fixMatrix` class is parametrized with a `CMatrix` that, for example, assumes values of type `float` and an index type `int`.

```
// Excerpt from k9/a3/divmat.cpp
int main() {
    fixMatrix<CMatrix<float,int> > MC(5,7);
    cout << " CMatrix " << endl;

    // fill rectangle
    for(int i = 0; i < MC.Rows(); ++i)
      for(int j = 0; j < MC.Columns(); ++j)
          // application of operator()():
          MC(i,j) = i + float(j/100.);

    // display rectangle
    for(int i = 0; i < MC.Rows(); ++i) {
       for(int j = 0; j < MC.Columns(); ++j)
          cout << MC(i,j) << ' ';
       cout << endl;
    }
    // ... (main() continued)
```

9.3.2 FORTRAN memory layout

The class for FORTRAN memory layout differs only by the kind of address calculation:

```
template<class ValueType, class IndexType>
class FortranMatrix : public MatrixSuperClass<ValueType,
                                              IndexType> {
  public:
   FortranMatrix(IndexType r, IndexType c,
          typename FortranMatrix::ContainerType& C)
   : MatrixSuperClass<ValueType, IndexType>(r,c,C) {
   }

   IndexType howmany() const {
       return this->Rows()*this->Columns();
   }

    /*In the address calculation, rows and columns are exchanged in contrast to the
      CMatrix class:
    */
   ValueType& where(IndexType r, IndexType c) const {
      assert(r < this->Rows() && c < this->Columns());
      return this->C[c * this->Rows() + r];
   }
};
```

A simple example shows the application:

```
fixMatrix<FortranMatrix<float, int> > MF(5,7);
// and so on, as above in the C matrix layout
```

9.3.3 Memory layout for symmetric matrices

There are several differences between this and the two previous classes: the constructor checks equality of numbers of rows and columns; the address and memory requirement calculations also differ.

```
template<class ValueType, class IndexType>
class symmMatrix
: public MatrixSuperClass<ValueType, IndexType> {
  public:
   symmMatrix(IndexType r, IndexType c,
         typename symmMatrix::ContainerType& C)
   : MatrixSuperClass<ValueType, IndexType>(r,c,C) {
      assert(r == c);    // matrix must be quadratic
   }

   // reduced memory consumption thanks to symmetry
   IndexType howmany() const {
       return this->Rows()*(this->Rows()+1)/2;
   }

   // the symmetry is exploited
   ValueType& where(IndexType r, IndexType c) const {
      assert(r < this->Rows() && c < this->Columns());
      if (r <= c) return this->C[r + c*(c+1)/2];
      else        return this->C[c + r*(r+1)/2];
   }
};
```

In the example, only one half-triangle of the matrix, including the diagonal, is equipped with values; nothing further is provided by the available memory. The subsequent display shows the complete matrix as a square where, obviously, the elements mirrored at the diagonal are equal.

```
// Example of a symmetric matrix, excerpt from k9/a3/divmat.cpp
    fixMatrix<symmMatrix<float, int> > MD(5,5);
    cout << "\n symmMatrix " << endl;

    // fill triangle
    for(int i = 0; i < MD.Rows(); ++i)
```

```
  for(int j = i; j < MD.Columns(); ++j)
      MD(i,j) = i + float(j/100.);

// output square
for(int i = 0; i < MD.Rows(); ++i) {
   for(int j = 0; j < MD.Columns(); ++j)
      cout << MD(i,j) << ' ';
   cout << endl;
}
```

9.4 Sparse matrices

A sparse matrix is one whose elements are nearly all zero. Sparse matrices find their application in simulation calculations of large networks in which mainly neighboring nodes are connected to each other. Examples include road networks, local and worldwide computer networks, telephone networks, compound systems for supplying the population with electricity, gas, and water, and many more. A characteristic feature of all these networks is their large number of nodes.

A matrix M may, for example, represent a road network in which the element M_{ij} contains the distance in kilometers between town i and town j. By convention, a value $M_{ij} = 0, (i \neq j)$ shall mean that no direct connection between towns i and j exists. A direct connection in this sense is a connection that connects exactly two towns. A road that touches several towns is therefore not considered as a direct connection between starting point and end point, but as a compound connection composed of direct connections. When one-way roads or direction-dependent routes play a role, $M_{ij} \neq M_{ji}$ may hold, so that M is not necessarily symmetric.

The fact that towns are directly connected with neighboring towns, but that there are barely any *direct* connections between distant towns, leads to the effect that the elements near the matrix diagonal are mostly not equal to 0. The ratio of the number of elements not equal to 0 and the total number of elements in the matrix is called the occupation rate. The occupation rate of a high-voltage network for energy supply, for example, is approximately $\frac{5\pm2}{N}$, where N is the number of network nodes and N^2 the number of matrix elements.

Network nodes	**Matrix elements**	**of which $\neq 0$**	**Occupation rate in %**
100	10 000	500	5
1 000	1 000 000	5 000	0.5
10 000	100 000 000	50 000	0.05

Table 9.1: Typical occupation rate in sparse matrices.

With 100 nodes, the matrix would have 10 000 elements, of which only about 500 would be not equal 0 (= 5%). Table 9.1 gives an idea of the dependency of the occupation rate on the number of nodes. It is obvious that it would be a waste of main and mass storage to store all the zeros. Therefore, typically only the non-zero elements are stored, together with an index pair (i, j) for identification.

Which abstract data type is best suited for storage of a sparse matrix? Imagine a column as a map which via a `long` index returns a `double` value. A matrix could then be a map which via a `long` index returns a row. Thus, a sparse matrix of double elements could be described quite simply as:

```
// k9/a4/sparse1.cpp
#include<map>
#include<iostream>
using namespace std;

// matrix declaration
typedef map<long, double> doubleRow;
typedef map<long, doubleRow> SparseMatrix;

/*The first index operator applied to a SparseMatrix returns a row on which the second
   index operator is applied, as shown in the program:
*/

int main() {
    SparseMatrix M;           // see declaration above
    M[1][1] = 1.0;
    M[1000000][1000000] = 123456.7890;
    cout.setf(ios::fixed);
    cout.precision(6);
    cout << M[1][1] << endl;                      // 1.000000
    cout << M[1000000][1000000] << endl;          // 123456.789000
    cout << "M.size() :" << M.size() << endl;     // 2

    /*Unfortunately, this very simple form of a sparse matrix has a couple of 'minor
       blemishes.' Access to a not yet defined element creates a new one:
    */
    cout << M[0][0] << endl;
    cout << "M.size() :" << M.size() << endl;     // 3

    /*This is not desirable, since the point is saving storage space. The next flaw is the
       uncontrolled access to unwanted positions, once again with the effect of generating
       additional elements:
    */
    cout << M[-1][0] << endl;                     // index error
    cout << "M.size() :" << M.size() << endl;     // 4
}
```

The maximum index cannot be defined anyway, because it is given by the number range of `long`. It would, however, be desirable to have a matrix which did not have these properties and which ensured that elements of value 0 did not contribute to memory consumption. Therefore, a different approach is presented which, however, requires more effort.

Here, access to the elements is carried out in a matrix via a pair of indices – row and column. Thus, an index pair constitutes the key for which the value of the matrix element is sought. This is a typical application of an associative container; thus, the classes `map` of the STL and `HMap` of Chapter 7 would be suitable, but in a different way than described above.

The following solution works with both kinds of map container, controlled by a compiler switch, but the second container is faster. Obviously, accessing an element of an associative container is slower when compared to a simple C array. This is the price that has to be paid for being able, for example, to represent a 1 000 000 000 × 1 000 000 000 matrix on a small PC and calculate with it, provided that the occupation rate is very, very small.

An example of the usage of a sparse matrix is shown in the following program segment in which a matrix with ten million rows and columns, that is, 10^{14} (fictitious) elements is defined. Control of whether the underlying container is to be taken from the STL is exercised by the switch `STL_map` which takes effect in the file *sparmat.h*. If the line is commented out using //, the `HMap` container of Chapter 7 is used.

```
// k9/a4/main.cpp
#include<iostream>
//#define STL_map
#include"sparmat.h"   // class sparseMatrix, see below
using namespace std;

// example of a very big sparse matrix
int main() {
    // ValueType double, IndexType long
    sparseMatrix<double, long> M(10000000,10000000);

    // Documentation
    cout << "matrix with "
         << M.rows()                     // 10000000
         << " rows and "
         << M.columns()                  // 10000000
         << " columns" << endl;

    // occupy some elements
    M[999998][777777]   = 999998.7777770;
    M[1][8035354]       = 123456789.33970;
    M[1002336][0]       = 444444444.1111;
    M[5000000][4900123] = 0.00000027251;

    // display of two elements
    cout.setf(ios::fixed|ios::showpoint);
    cout.precision(8);
    cout << "M[1002336][0]       = "
         <<  M[1002336][0]      << endl;

    cout << "M[5000000][4900123] = "
         <<  M[5000000][4900123]  << endl;
```

The output is

```
M[1002336][0]           = 444444444.11110002
M[5000000][4900123]     = 0.00000027
```

The small deviations with respect to the above assignments result from formatting with `precision(8)`. Besides row and column number, it is also possible to output the number of non-zero elements:

```
cout << "Number of non-zero elements = "
     << M.size() << endl;

cout << "max. number of non-zero elements = "
     << M.max_size() << endl;
```

To satisfy the need to output all non-zero elements of the sparse matrix for display or storage, the `sparseMatrix` class should provide forward iterators:

```
cout << "Output all non-zero elements via iterators\n";
sparseMatrix<double, long>::iterator temp = M.begin();
while(temp != M.end()) {
    cout << "M["   << M.Index1(temp)     // i
         << "]["   << M.Index2(temp)     // j
         << "] = " << M.Value(temp)      // value
         << endl;
    ++temp;
}
// ...
```

The above lines lead to the following display

```
Output all non-zero elements via iterators
M[1][8035354] = 123456789.33970000
M[5000000][4900123] = 0.00000027
M[1002336][0] = 444444444.11110002
   ⋮
```

The output is only ordered when the map container of the STL is chosen. In the above example, this is obviously not the case.

9.4.1 Index operator and assignment

Because of the selective storage of matrix elements, some peculiarities must be considered during the design, particularly of the index and assignment operators. A matrix element can stand both on the left-hand and on the right-hand side of an assignment. In both cases, it must be taken into account that the element might not yet exist in the container, that is, if it has not yet been assigned a value not equal to zero. Three cases must be distinguished (the matrix elements are to be of type `double`):

1. Matrix element as lvalue: `M[i][j] = 1.3;`
   In order to analyze it, the instruction must be broken down into its function parts:

   ```
   sparseMatrix::operator[](i).operator[](j).operator=(1.3);
   ```

   The first index operator checks the line index `i` for maintaining the limits, the second operator checks the column index `j`. Furthermore, the second index operator must supply an object that possesses an assignment operator to enter a `double` value into the associative container together with the indices. This object must have available all necessary information. When the `double` value is zero, however, *no* entry is to be made, but the element `M[i][j]` is to be deleted, provided it already exists.

   As usual in C++, some auxiliary classes are invented for the solution of this problem. The first class, named `Aux`, is the return type of the first index operator. The second index operator, which checks the column number, is the index operator of the `Aux` class. It returns an object of type `MatrixElement`, the second auxiliary class. The assignment operator of this object caters for the rest, as illustrated by the following lines:

   $$\underbrace{\texttt{sparseMatrix::operator[](i)}}\texttt{.operator[](j).operator=(1.3);}$$
   $$\underbrace{\texttt{Aux::operator[](j)}}\texttt{.operator=(1.3);}$$
   $$\texttt{MatrixElement::operator=(1.3);}$$

   On the surface, this highly flexible method of proceeding may seem costly. However, this cost must be compared with the insertion and search processes of the underlying container; then, the balance looks significantly better. Substituting the usual index operator `operator[]()` with the function operator `operator()()` brings no advantage.

2. Matrix element as rvalue: `double x = M[i][j];`
   In addition, the `MatrixElement` class needs an operator which converts an object of type `MatrixElement` into the appropriate value type, in this case `double`.

3. Matrix element on both sides: `M1[n][m] = M2[i][j];`, where `M1` and `M2` may be identical.
   The `MatrixElement` class needs a second assignment operator with the argument `const MatrixElement&`.

9.4.2 Hash function for index pairs

In this section, the file *sparmat.h* is presented which contains the classes and auxiliary classes discussed above. It is included via `#include` into a program which is designed to work with sparse matrices (see example on page 212). The file begins with some preprocessor directives for determining the underlying implementation.

```
// File k9/a4/sparmat.h, templates for sparse matrices
#ifndef SPARSEMATRIX_H
#define SPARSEMATRIX_H

// selection of implementation
#ifdef STL_map              // defined in main()
#include<map>
#include<cassert>
#else
#include"hmap.h"

/*If at this point the HMap container of Chapter 7 is chosen, a function for calculating the
  hash table addresses is needed. As opposed to the hash functions described up to now,
  not just one value, but two are used for the calculation. Therefore, the function operator
  of the PairHashFun class takes a pair as argument. The address calculation itself is
  simple, but sufficient for the examples in this book.
*/

template<class IndexType>  // int, long or unsigned
class PairHashFun {
  public:
        PairHashFun(long prime=65537)
        // Another prime number is possible.
        // for example, 2111 for smaller matrices.
        : tabSize(prime) {
        }

    // Address calculation with two values
    long operator()(
          const std::pair<IndexType, IndexType>& p) const {
       return (p.first + p.second) % tabSize;
    }

    long tableSize() const { return tabSize;}

  private:
    long tabSize;
};
#endif     // STL_map or not
```

9.4.3 Class MatrixElement

An element stored in a container has a determined type denoted in the STL by `value_type`. In this case, the `value_type` is a pair consisting of the key and the associated value, where the key itself is a pair of two indices. In the class described below, a pair of indices is defined as type `IndexPair`.

```
template<class ValueType, class IndexType,
         class ContainerType>
class MatrixElement {
  private:
    ContainerType& C;
    typename ContainerType::iterator I;
    IndexType row, column;

  public:
    typedef std::pair<IndexType, IndexType> IndexPair;
    typedef MatrixElement<ValueType, IndexType,
                              ContainerType>&  Reference;
```

/*The constructor initializes the private variables with all information that is needed. (Normally, the private objects are placed at the end of a class definition. For reasons of contextual consistency, this rule is sometimes not observed.) The container itself is located in the `sparseMatrix` class; here, the reference to it is entered. If the passed indices for row and column belong to an element not yet stored in the container, the iterator has the value `C.end()`.
*/

```
    MatrixElement(ContainerType& Cont,
                  IndexType r, IndexType c)
    : C(Cont), I(C.find(IndexPair(r,c))),
      row(r), column(c) {
      }

    ValueType asValue() const {
       if(I == C.end())
           return ValueType(0);
       else
           return (*I).second;
    }

    operator ValueType () const    {// type conversion operator
       return asValue();
    }
```

/*According to the definition of the sparse matrix, 0 is returned if the element is not present in the container. Otherwise, the result is the second part of the object of type `value_type` stored in the container. The type conversion operator fulfils the requirements of point 2 on page 214. The assignment operator (see point 1 on page 214) is structured in a slightly more complicated way.
*/

```
    Reference operator=(const ValueType& x) {
       if(x != ValueType(0)) {        // not equal to 0?
```

/*If the element does not yet exist, it is put, together with the indices, into an object of type `value_type` and inserted with `insert()`:
*/

```
          if(I == C.end()) {
              assert(C.size() < C.max_size());
              I = (C.insert(typename ContainerType
                     ::value_type(IndexPair(row,column), x))
                  ).first;
           }
           else (*I).second = x;
        }

        /*insert() returns a pair whose first part is an iterator pointing to the in-
           serted object. The second part is of type bool and indicates whether the in-
           sertion took place because no element with this key existed. This is, however,
           not evaluated here because, due to the precondition (I == C.end()), the
           second part must always have the value true. If, instead, the element already
           exists, the value is entered into the second part of the value_type object.

           If the value is equal to 0, then to save space the element is deleted if it existed.
        */
        else                         // x = 0
           if(I != C.end()) {
               C.erase(I);
               I = C.end();
           }
        return *this;
     }

     /*Point 3 on page 214 requires an assignment operator which in turn requires a
        reference to an object of type MatrixElement. When both the left- and right-
        hand sides are identical, nothing has to happen. Otherwise, as above, it has to be
        checked whether the value of the right-hand element is 0 or not. The resulting
        behavior is described together with the above assignment operator, so that here it
        is simply called:
     */
     Reference operator=(const Reference rhs) {
        if(this != &rhs) {      // not identical?
            return operator=(rhs.asValue());   // see above
        }
        return *this;
     }
};   // class MatrixElement
```

9.4.4 Class sparseMatrix

Depending on the selected implementation, the data types for the container and other aspects are set:

```
template<class ValueType, class IndexType>
class sparseMatrix {
   public:
     typedef std::pair<IndexType, IndexType> IndexPair;

// The switch STL_map controls the compilation:

#ifdef STL_map
     typedef std::map<IndexPair, ValueType,
             std::less<IndexPair> >       ContainerType;
#else
     typedef br_stl::HMap<IndexPair, ValueType,
                  PairHashFun<IndexType> > ContainerType;
#endif

     typedef MatrixElement<ValueType, IndexType,
                           ContainerType> MatrixElement;

  public:
    typedef IndexType size_type;

    /*The constructor initializes only the row and column information. The container is
      created by its default constructor, where in the case of hash implementation, the
      size of the container is given by the hash function object of type PairHashFun
      (see typedef above).
    */
  private:
    size_type rows_, columns_;
    ContainerType C;

  public:
    sparseMatrix(size_type r, size_type c)
    : rows_(r), columns_(c) {
    }

    /*The following list of methods, besides determining the number of rows and
      columns, provides the common container methods, which are not discussed in
      detail.
    */
   size_type rows()  const   { return rows_;}
   size_type columns() const { return columns_;}

   // usual container type definitions
   typedef typename ContainerType::iterator iterator;
   typedef typename ContainerType::const_iterator
                                      const_iterator;

   // usual container functions
   size_type size()       const { return C.size();}
   size_type max_size()   const { return C.max_size();}
```

```
iterator begin()             { return C.begin();}
iterator end()               { return C.end();}

const_iterator begin() const { return C.begin();}
const_iterator end()   const { return C.end();}

void clear() { C.clear();}

class Aux {
  public:
    Aux(size_type r, size_type maxs, ContainerType& Cont)
    : Row(r), maxColumns(maxs), C(Cont) {
    }

    /*After checking the number of columns, the index operator of Aux returns a
      matrix element which is equipped with all the necessary information to carry
      out a successful assignment.
    */
    MatrixElement operator[](size_type c) {
        assert(c >= 0 && c < maxColumns);
        return MatrixElement(C, Row, c);
    }

  private:
    size_type Row, maxColumns;
    ContainerType& C;
};

/*The index operator of the sparseMatrix class returns the auxiliary object de-
  scribed on page 214, whose class is defined as nested inside sparseMatrix.
*/
Aux operator[](size_type r) {
   assert(r >= 0 && r < rows());
   return Aux(r, columns(), C);
}

/*Up to this point, from a functionality point of view, the sparseMatrix
  class is sufficiently equipped. However, to avoid writing such horrible things as
  '(*I).first.first' for accessing the elements, some of the following auxil-
  iary functions determine the indices and associated values of an iterator in a more
  readable way. Their application can be seen in the example on page 213.
*/
size_type Index1(const_iterator& I) const {
   return (*I).first.first;
}

size_type Index2(const_iterator& I) const {
   return (*I).first.second;
}
```

```
   ValueType Value(const_iterator& I) const {
      return (*I).second;
   }
};          // class sparseMatrix
#endif      // file sparmat
```

From the point of view of the information needed in the auxiliary functions, it is not necessary to formulate these functions as member functions. It would also be possible to create template functions which are not members. These, however, would need an extra parameter to determine the type of the index or the values, thus the first way is followed.

9.4.5 Run time measurements

Owing to its more complicated storage, access to an element of a sparse matrix takes significantly longer than access to elements of the matrices discussed in the previous sections.

Figure 9.1 shows how the access time to a matrix element depends on the number N of elements already in the container. The access time depends on the kind of computer, the operating system, and the compiler and its settings. The measurements were carried out using a 233 MHz Pentium PC, the egcs-1.0.2 C++-Compiler and the Linux operating system.

The dot sequences show the trend. The round dots of nearly constant access time apply to the implementation of the `sparseMatrix` class with a `HMap` container; the ascending sequence of square dots shows the linear dependency of the access time from the logarithm of the number N of already stored elements of the sorted map container of the STL.

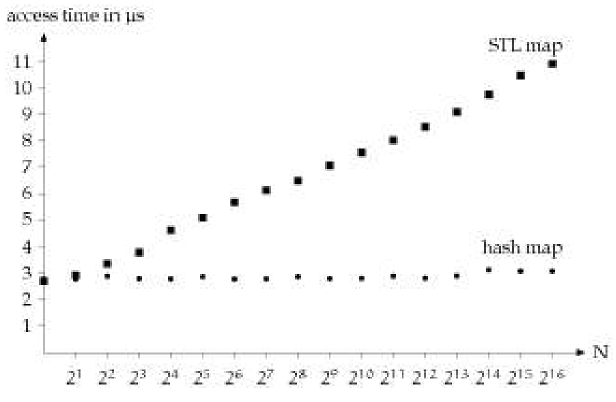

Figure 9.1: Access times for elements of a sparse matrix.