Exercises for the Lecture Series

"Object-Oriented Programming for Scientific Computing"

Ole Klein
ole.klein@iwr.uni-heidelberg.de

To be handed in on 16. 06. 2015 before the lecture

EXERCISE 1 STL CONTAINER TYPES

In the lecture the different types of containers of the STL have been discussed. Each type offers certain guarantees and is ideally suited for specific use cases.

Here is a list of scenarios, what type of container would you use and why?

1. You are writing a finite difference discretization of the Laplace equation, and you are using a structured grid. For each grid node, the solution at this point is stored. All nodes of the grid are numbered consecutively and their number is known a priori.

   Which type of container is suitable for storing the node values?

2. You are writing a Quicksort sorting algorithm, and you want to store the pivot elements in a stack structure to save the recursive function calls.

   Which type of container is well suited to save the pivot elements and which to store the data itself?

3. When implementing direct solvers for sparse matrices, it is common to first minimize the bandwidth of the matrix in order to reduce the storage requirements of the resulting LU decomposition. One method for this is the Cuthill-McKee algorithm. In the course of this algorithm elements have to be buffered in a FIFO (first-in, first-out) data structure. The order of the elements in the FIFO does not change.

   What type of container would you use as a FIFO?

4. In your finite difference program, the solution is defined a priori on a part of the boundary. Nodes in this part of the boundary (Dirichlet nodes) aren't real degrees of freedom and must be treated differently when assembling the Matrix. The number of these nodes is small compared to the total number of nodes.

   You want to dynamically read the list of Dirichlet nodes and their solution values from a configuration file and make them accessable node by node. When solving linear PDEs the limiting factor is typically the memory, as you want to solve very big problems.

   In what container you would store the indices of Dirichlet nodes and their values?

10 Points

EXERCISE 2 MATRIX IITERATORS

Up to now all matrices appearing in the lectures and the exercises were dense matrices, i.e. we did not impose any constraints on their entries. In Scientific Computing we often have additional information about the internal structure of the matrices, and this structure can be used to create efficient algorithms. Typically the matrices are sparse, i.e. most entries are zero, for example banded matrices.

In this situation it is natural to only store the entries that are non-zero by using optimized data structures. There are may different encodings for the matrix contents that exploit the sparse structure, each with advantages and drawbacks in specific situations. Since algorithms should often work with more than one of these approaches, and depending on the algorithm also for dense matrices, an abstract data access scheme has to be used. A suitable abstraction already known from the STL is the iterator concept.

1. Extend the templatized `MatrixClass` from exercise sheet 7. If you did not succeed on that exercise, you may instead use the original `double` variant you can find on the website. Since this is a dense matrix, only the iterator concept has to be implemented. In a second step the matrix could be changed to a sparse matrix, but that is too much for the scope of this exercise.

2. We introduce two different types of iterator:

   Row iterators:

   ```
   class RowIterator
   {
     RowIterator& operator++();     // move to next row
     bool operator==() const;       // comparison of iterators
     Row& operator*();              // access to current row
     const Row& operator*() const; // as above, but const
     unsigned int row() const;      // number of current row
   };
   ```

   Column iterators:

   ```
   class ColIterator
   {
     ColIterator& operator++();     // move to next entry
     bool operator==() const;       // comparison of iterators
     T& operator*();                // access to current entry
     const T& operator*() const;    // as above, but const
     unsigned int col() const;      // number of current entry
   };
   ```

3. The `MatrixClass` is now a container of `Row`s, which are in turn containers for row entries.

4. The matrix gives access to the row iterators, which in turn give access to a single row. The row gives access to the column iterators, which in turn give access to the individual entries.

5. Write `begin()` and `end()` methods for the matrix, both returning a `RowIterator`. `begin()` should point to the first non-empty row (here: first row), and `end()` to the position after the last row.

6. The class `Row` is also new and has to be implemented. The minimum requirements are `begin()` and `end()` methods that both return a `ColIterator`. `begin()` should point to the first non-zero entry (here: first entry), and `end()` to the position after the last column.

7. Implement all mentioned classes as *nested classes* of `MatrixClass`. The interface is given above, but the details are up to you. You can introduce additional private methods and data members for the classes, and you will most likely have to change methods that existed before.

8. Reimplement all methods of `MatrixClass` that can use the iterator concept, i.e. the methods `Print`, `operator*=` and `operator+=`. The last method would normally have to take into account that the second matrix might have a different sparsity pattern, but you can ignore this here. Test your implementation with the same file as on exercise sheet 7.

10 Points