Exercises for the Lecture Series

"Object-Oriented Programming for Scientific Computing"

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

To be handed in on 14. 07. 2015 before the lecture

This is the last exercise sheet of the lecture series. The points of the exercises presented here do not count towards the total amount of points needed on the exercise sheets, they are bonus points that you may get if you require additional points for the admission to the exam.

EXERCISE 1 TYPE ERASURE (BONUS EXERCISE)

Generic programming with templates offers many opportunities to write flexible, highly efficient code. Programs with dynamic polymorphism, however, are usually easier to develop and maintain. They also offer greater flexibility because they allow interchangeability at runtime.

In practice, a program consists of performance-critical areas and areas where the cost of dynamic polymorphism can be neglected. Therefore, a mechanism for hiding the static polymorphism may be useful, either in certain areas or at a certain level of abstraction, to spare the user the unnecessary template constructs.

The method of `type erasure` provides these options. Examples of libraries that are using this method are `boost::any` or `adobe::any_iterator`.

Example:

```
#include <list>
#include <boost/any.hpp>

struct Foo {};

int main() {
    // some arbitrary data
    int i;
    double d;
    Foo f;

    // store any data
    std::list<boost::any> many;

    many.push_back(boost::any(i));
    many.push_back(boost::any(d));
    many.push_back(boost::any(f));

    return 0;
}
```

We will now attempt to program such a construction on our own. The goal is an abstract stack that stores `int` values:

```
class Stack {
    void push_back(int); // append an element
    void pop_back();     // delete last element
    int back();          // read last element
};
```

It should be possible to initialize this abstract stack with any container that provides these three methods (e.g. `std::vector`, `std::deque`, `std::list`).

The construction uses the following trick. The `Stack` has to store a pointer to the actual container. However, the container type is not a priori known, therefore a virtual interface `StackInterface` is

introduced for the container. This interface contains exactly the methods of the stack, but they are all purely virtual methods. The `Stack` now contains a pointer to `StackInterface` and all the calls to `push_back`, etc. are forwarded to the appropriate method of the internal `StackInterface` object.

This virtual interface leads to a second problem. One cannot assign a pointer to e.g. `std::vector` to the `StackInterface` pointer. In order to accomplish this, one introduces a templatized wrapper class `StackImpl` that satisfies the virtual interface `StackInterface`. The template parameter of `StackImpl` now corresponds to the actual implementation (e.g. `std::vector`). The class stores a pointer to an object of this implementation and redirects all virtual function calls statically to this object.

Note : As an alternative to the implementation of `StackImpl` *the Curiously Recurring Template Pattern* [†] *may also be used.*

One may now relatively easily program the classes so that the following call could be used:

```
Stack s = StackImpl(new std::vector);
```

In order to hide all the constructions containing `StackInterface` and `StackImpl` from the user, the `Stack` finally receives a templatized constructor which receives an arbitrary container `T` that statically satisfies the interface as argument. In this constructor, a dynamic wrapper `StackImpl<T>` is created on the heap and its pointer is stored as a `StackInterface*` member of the `Stack`.

In the end, we arrive at a class `Stack` that leaves no trace of the type of the container that is used internally. Objects of this class may now be stored in the same container even if they use completely different storage mechanisms inside, and they may be used in functions that don't contain templates at all.

1. Write a test program that checks that your `type erasure` construction works.
2. Implement `type erasure` for the example class `Stack` above.

10 Bonus Points

EXERCISE 2 SPARSE MATRICES

The matrices in Scientific Computing often have very many entries that are zero, and the non-zero entries follow a pattern. This sparse matrix structure is a direct consequence of the discretization. There are very efficient algorithms to solve equations that contain such matrices, and the Gauss-Seidel method we had on the exercise sheets is a simple example of such a method.

These algorithms can only be implemented in an efficient way (both computing time and memory consumption) if the special structure is considered when storing and accessing the matrix elements.

In this exercise we want to implement a simple variant of such a sparse matrix.

- We use the Compressed Row Storage ansatz.
- This is an actual application of the abstract matrix interface from sheet 9: Iterating over the matrix with `RowIterator` and `ColIterator`, as well as the corresponding `begin` and `end` methods, allows skipping all entries that are zero.
- Entries which are zero won't be saved and will be skipped by the iterators (as they don't contribute to matrix operations). Algorithms that use the interface have to query the current row and/or column from the iterators to handle the entries that are returned.
- Since only the non-zero entries are stored, it is relevant to store their number in the matrix. This happens in the constructor:

```
SparseMatrix(unsigned int rows, unsigned int cols, unsigned int nonZeros);
```

[†] `https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern`

- The data structure looks like this (pseudo code):

```
template<typename T>
SparseMatrix {
    T data[nonZeros];
    unsigned int column[nonZeros];
    unsigned int rowOffset[rows];
};
```

  - The data is stored in rows in `data`.
  - For each non-zero entry in `data`, the corresponding entry in `column` stores in which column it is.
  - The `rowOffset` vector stores for each row the first index in `data` (and `column`) that contains an entry from the row.

  Example (Matrix from exercise sheet 10):

```
rows: 3
cols: 3
nonZeros: 7
data: 2, 1, 1, 3, 2, 1, -4
column: 0, 1, 0, 1, 2, 1, 2
rowOffset: 0, 2, 5
```

- Implement the sparse data structure above. If you have already used the iterators for the methods of your `MatrixClass`, you should only need to modify the data container and the iterators, while the rest of the matrix implementation can be copied over.
- Test your implementation with one of the two matrix algorithms from exercise sheet 10, using the Gauss-Seidel algorithm if you have it.

Note that many algorithms, the Gauss-Seidel method included, treat diagonal entries in a special manner. Therefore it often makes sense to store them apart from the other entries in a fourth vector `T diag[rows]` *and leave them out in* `data`*. This allows the introduction of dedicated methods for the access to diagonal entries. Since this variant makes the implementation of the iterators more complicated, it is not part of this exercise, but you may implement it instead of the version given above if it interests you.*

10 Bonus Points