# Course $C^{++}$, Exercise Number 10

Date: 11.05.2012 + two weeks

Implementing Rubik cube for arbitrary size is not easy. One way of doing it, is by analyzing the three dimensional structure, determine on which coordinate a face is situated (a 3-vector), and to determine in which direction it faces (another 3-vector). One could implement the rotations through linear algebra operators. This is possible, but complicated, and unlikely to be efficient.

We will do it in another way: We put all faces in a single, flat vector, which will have size $6S^2$, where $S$ is the size of the cube. We implement indexing operators (I did this already), so that the faces can be accessed as `cube [ color ] [ unsigned int ] [ unsigned int ]`. In order to determine which faces are moved by which rotation, we will use tables and switch statements. This is not nice, but it is the easiest way, and also the most efficient.

First read the explanation on the homepage, of how rotations are encoded. Remember that the implementation must work for arbitrary dimension.

Download the files, and make sure that you understand how the indexing works, because it is a useful technique.

1. Implementing the cube rotations is rather hard. Let's first do an easy thing: Add a method

   ```
   static void
   rotate( int d, color& c1, color& c2, color& c3, color& c4 );
   ```

   This rotates the colors c1,c2,c3,c4 over a distance specified by $d$. If $d = 1$, the colors are rotated one step right. If $d = -1$, the colors are rotated one step left. If $d = 2$, then c1,c3 are exchanged, and c2,c4 are exchanged. If $d = 0$, then nothing happens. If you want, you can also make it work for the other integers, because the effect of rotating over distance $d$ is always the same modulo 4.

2. Now we need to implement rotations of form $S^i_j$. Special attention needs to be given to the situation where $j = 0$ or $j = S-1$. In that case, we need to rotate a complete surface. Otherwise, we have to rotate only middle faces.

   Add a method

```
void rotate_surface_only( side s, int dist );
```

to cube. Since you are member of `cube`, you can obtain a `side_index`, and use this side index for the rotation:

On a 4-cube, the code should have the following effect: (But of course, you have to write `for` loops, because the code must work for every $S$.)

```
side_index p = (*this) [ s ];
rotate( dist, p[0][0], p[3][0], p[3][3], p[0][3] );
rotate( dist, p[1][0], p[3][1], p[2][3], p[0][2] );
rotate( dist, p[0][1], p[2][0], p[3][2], p[1][3] );
rotate( dist, p[1][1], p[2][1], p[2][2], p[1][2] );
```

It is a bit tricky, to get the indices right in the general case, but it should be possible. If $S$ is an odd number, then the middle surfaces don't move.

3. A rotation $S^i_j$ also affects the neighbours of the surface $S$. For example, rotating $L$ on a 4-cube, should have the following effect (in addition to rotating the left surface):

```
rotate( 1, (*this)[front][0][0], (*this)[up][0][0],
           (*this)[back][3][3], (*this)[down][0][0] );
rotate( 1, (*this)[front][1][0], (*this)[up][1][0],
           (*this)[back][2][3], (*this)[down][1][0] );
rotate( 1, (*this)[front][2][0], (*this)[up][2][0],
           (*this)[back][1][3], (*this)[down][2][0] );
rotate( 1, (*this)[front][3][0], (*this)[up][3][0],
           (*this)[back][0][3], (*this)[down][3][0] );

Rotating $ D^{-1}_2 $ on a 4 cube has no effect on the
down surface itself, but it has the following effect on the
middle surfaces:
\begin{verbatim}
rotate( -1, (*this)[front][2][0], (*this)[left][2][0],
            (*this)[back][2][0], (*this)[right][2][0] );
rotate( -1, (*this)[front][2][1], (*this)[left][2][1],
            (*this)[back][2][1], (*this)[right][2][1] );
rotate( -1, (*this)[front][2][2], (*this)[left][2][2],
            (*this)[back][2][2], (*this)[right][2][2] );
rotate( -1, (*this)[front][2][3], (*this)[left][2][3],
            (*this)[back][2][3], (*this)[right][2][3] );
```

Finding the positions that must be rotated, is very tricky. The best solution is to define, using `switch` statements, for every surface, the following:

(a) Its four neighbouring surfaces (in positive orientation).
(b) For each neigbouring surface, where the first layer starts. (Two **unsigned ints**.)
(c) For each neigbouring surface, how one should move to the next layer. (Two **ints**.)
(d) For each neigbouring surface, how one moves within the same layer. (Two **ints**.)

You can use switch statements, or tables. Once you have the tables, writing the actual code is not hard anymore. (I tried it.) Complete the method

```
      void rotate_neighbours( side s, int dist, unsigned int layer );
```

4. Now you can add the method

```
void cube::rotate( const rotation& rot )
{
   if( rot. layer == 1 )
      rotate_surface_only( rot. s, rot. dist );

   if( rot. layer == size )
   {
      side opposite = up;
      switch( rot. s )
      {
      case left:      opposite = right; break;
      case right:     opposite = left; break;
      case up:        opposite = down; break;
      case down:      opposite = up; break;
      case front:     opposite = back; break;
      case back:      opposite = front; break;
      }
      rotate_surface_only( opposite, - rot. dist );
   }

   rotate_neighbours( rot. s, rot. dist, rot. layer );
}
```

that is able to carry out a rotation.

5. It is now easy to complete the method:

```
   void cube::rotate( const sequence& seq );
```

6. Check that the sequence

$$L_{1,3} \cdot U \cdot L_3^{-1} \cdot U^{-1} \cdot L_1^{-1} \cdot U \cdot L_3 \cdot U^{-1} \cdot L_3^{-1}$$

works as specified in the example. The sequence

$$(R \cdot D^{-1} \cdot R^{-1} \cdot D)^2 \cdot U \cdot (D^{-1} \cdot R \cdot D \cdot R^{-1})^2 \cdot U^{-1}$$

should rotate two corners. (And nothing else).

If you want, use can use graphics to plot the cube. This works in Room 7. You have to switch all the `#if`s that are now switched of. In addition, you have to use the modified Makefile.

If the compilers complains about the use of the initialization lists, you have add the option `std=c++0x`.