Exercises week 2

Klaas Isaac Bijlsma
s2394480

David Vroom
s2309939

November 27, 2017

Exercise 11

Learn to appreciate catching references when throwing exceptions

A simple class `Object` is made. It has a data member `d_name` that stores an internal name. If an object is made via the copy constructor, 'copy' is added to this internal name. The constructor, copy constructor and destructor print what they did together with the internal name. A function `hello()` is added that says hello and prints the internal name.

object/object.h

```cpp
#ifndef INCLUDED_OBJECT_
#define INCLUDED_OBJECT_

#include <string>

class Object
{
    std::string d_name;

    public:
        Object(std::string const &name);         // 1
        Object(Object const &other);             // 2
        ~Object();
        void hello();
};

#endif
```

object/object.ih

```
#include "object.h"
#include <iostream>
#include <string>

using namespace std;
```

object/destructor.cc

```
#include "object.ih"

Object::~Object()
{
    cout << "Destructed '" << d_name << "'\n";
}
```

object/hello.cc

```
#include "object.ih"

void Object::hello()
{
    cout << "Hello, this is '" << d_name << "'\n";
}
```

object/object1.cc

```
#include "object.ih"

Object::Object(string const &name)
:
    d_name(name)
{
    cout << "Constructed '" << d_name << "'\n";
}
```

object/object2.cc

```
#include "object.ih"

Object::Object(Object const &other)
:
    d_name(other.d_name + " (copy)")
{
    cout << "Copy constructed '" << d_name << "'\n";
}
```

Below a main function (`main1.cc`) is shown, in which within a try block, an object of the class `Object` is made. This object is then thrown. The exception handler catches an object of the class `Object` (by value). The output of the program is given below the code of main1. We see that the object is properly constructed and says hello. Then when it is thrown, first a copy is made and the original object is destructed. The copy is passed to the exception handler. Here an additional copy is made, because it receives the object by value. Therefore, within the exception handler, the copy of the copy of the object says hello.

main1.cc

```
#include "object/object.h"
#include <iostream>
using namespace std;

int main()
{
    try
    {
        Object object{ "object" };
        object.hello();
        throw object;
    }
    catch (Object caughtObject)
    {
        cout << "Caught exception\n";
        caughtObject.hello();
    }
}
```

Output of main1.cc

```
Constructed 'object'
Hello, this is 'object'
Copy constructed 'object (copy)'
Destructed 'object'
Copy constructed 'object (copy) (copy)'
Caught exception
Hello, this is 'object (copy) (copy)'
Destructed 'object (copy) (copy)'
Destructed 'object (copy)'
```

The following main function (`main2.cc`) does the same as the previous, except that the exception handler catches *a reference* to an object of the class `Object`. From the output we see that no second copy is made. This is more efficient and therefore exception handlers should catch references to objects.

main2.cc

```
#include "object/object.h"
#include <iostream>
using namespace std;

int main()
{
    try
    {
        Object object{ "object" };
        object.hello();
        throw object;
    }
    catch (Object &caughtObject)
    {
        cout << "Caught exception\n";
        caughtObject.hello();
    }
}
```

Output of main2.cc

```
Constructed 'object'
Hello, this is 'object'
Copy constructed 'object (copy)'
Destructed 'object'
Caught exception
Hello, this is 'object (copy)'
Destructed 'object (copy)'
```

In the previous two programs, we saw that a copy of the object is thrown. This is because the original object is a local object that only lives inside the try block. The same is true when a reference to an object is thrown, as can be seen from the output that the following code produces:

main3.cc

```
#include "object/object.h"
#include <iostream>
using namespace std;

int main()
{
    try
    {
        Object object{ "object" };

        Object &ref = object;
        ref.hello();
        throw ref;
    }
    catch (Object &caughtObject)
    {
        cout << "Caught exception\n";
        caughtObject.hello();
    }
}
```

Output of main3.cc

```
Constructed 'object'
Hello, this is 'object'
Copy constructed 'object (copy)'
Destructed 'object'
Caught exception
Hello, this is 'object (copy)'
Destructed 'object (copy)'
```

The following main function (`main4.cc`) has two exception levels. In the inner level, an object of the class `Object` is thrown and caught as a reference. Then it is rethrown to a more shallow level where it is again caught as a reference. From the shown output, we conclude that '`throw;`' results in throwing the currently available exception and not a copy of that exception.

main4.cc

```
#include "object/object.h"
#include <iostream>
using namespace std;

int main()
{
    try
    {
        try
        {
            Object object{ "object" };
            object.hello();
            throw object;
        }
        catch (Object &caughtObject)
        {
            cout << "Caught exception in inner block\n";
            caughtObject.hello();
            throw;
        }
    }
    catch (Object &caughtObject)
    {
        cout << "Caught exception in outer block\n";
        caughtObject.hello();
    }
}
```

Output of main4.cc

```
Constructed 'object'
Hello, this is 'object'
Copy constructed 'object (copy)'
Destructed 'object'
Caught exception in inner block
Hello, this is 'object (copy)'
Caught exception in outer block
Hello, this is 'object (copy)'
Destructed 'object (copy)'
```

Exercise 12

Exercise 13

Learn to create an exception safe class

We modified the following code,

main.cc

```
#include "main.ih"

int main(int argc, char **argv)
try
{
    cout << "3 x 3 matrix filled with zeros\n";
    Matrix mat(3, 3);
    show(cout, mat) << '\n';

    cout << "4 x 4 identity matrix\n";
    show(cout, Matrix::identity(4)) << '\n';

    cout << "Changing the 3 x 3 matrix into a 3 x 4 matrix filled with 1..12"
         << '\n';
    mat = Matrix{
                   { 1,  2,  3},
                   { 4,  5,  6},
                   { 7,  8,  9},
                   {10, 11, 12},
               };

    show(cout, mat) << '\n';

    cout << "Transposing the above matrix:\n";
    show(cout, mat.transpose()) << '\n';
}
catch (...)
{}
```

matrix/matrix.h

```
#ifndef INCLUDED_MATRIX_
```

```cpp
#define INCLUDED_MATRIX_

#include <iosfwd>
#include <initializer_list>

class Matrix
{
    size_t d_nRows = 0;
    size_t d_nCols = 0;
    double *d_data = 0;                     // in fact R x C matrix

    public:
        typedef std::initializer_list<std::initializer_list<double>> IniList;

        Matrix() = default;
        Matrix(size_t nRows, size_t nCols);         // 1
        Matrix(Matrix const &other);                // 2
        Matrix(Matrix &&tmp);                       // 3
        Matrix(IniList inilist);                    // 4

        ~Matrix();

        Matrix &operator=(Matrix const &rhs);
        Matrix &operator=(Matrix &&tmp);

        double *row(size_t idx);
        double const *row(size_t idx) const;

        size_t nRows() const;
        size_t nCols() const;
        size_t size() const;            // nRows * nCols

        static Matrix identity(size_t dim);

        Matrix &tr();                   // transpose (must be square)
        Matrix transpose() const;       // any dim.

        void swap(Matrix &other) noexcept;

        double &at(size_t rowIdx, size_t colIdx);   // NEW


    private:
        double &el(size_t row, size_t col) const;
        void transpose(double *dest) const;
};

inline double *Matrix::row(size_t row)
{
    return &el(row, 0);
}

inline double const *Matrix::row(size_t row) const
{
    return &el(row, 0);
}

inline size_t Matrix::nCols() const
{
    return d_nCols;
}

inline size_t Matrix::nRows() const
{
    return d_nRows;
}

inline size_t Matrix::size() const
{
    return d_nRows * d_nCols;
}

inline double &Matrix::el(size_t row, size_t col) const
{
    return d_data[row * d_nCols + col];
}

#endif
```

matrix/at.cc

```cpp
#include "matrix.ih"

double &Matrix::at(size_t rowIdx, size_t colIdx)
{
    try
    {
        if (rowIdx >= d_nRows)
            throw "Row index exceeded";
        if (colIdx >= d_nCols)
            throw "Column index exceeded";
        return el(rowIdx, colIdx);
    }
    catch (char const *message)
    {
        cerr << "Exception: " << message << '\n';
        throw;
    }
}
```

matrix/identity.cc

```cpp
// This function offers the basic guarantee. If it cannot make an identity
// matrix, the allocated memory is returned (handled in the corresponding
// constructor).

#include "matrix.ih"

// static
Matrix Matrix::identity(size_t dim)
try
{
    Matrix ret(dim, dim);

    for (size_t idx = 0; idx != dim; ++idx)
        ret.el(idx, idx) = 1;

    return ret;
}
```

```
catch (...)
{
    cerr << "Could not make identity matrix\n";
    throw;
}
```

matrix/matrix1.cc

```
#include "matrix.ih"

Matrix::Matrix(size_t nRows, size_t nCols)
try
:
    d_nRows(nRows),
    d_nCols(nCols),
    d_data(0)
{
    d_data = new double[size()]();
}
catch (...)
{
    cerr << "Memory allocation failed\n";
    delete[] d_data;
}
```

matrix/matrix2.cc

```
#include "matrix.ih"

Matrix::Matrix(Matrix const &other)
try
:
    d_nRows(other.d_nRows),
    d_nCols(other.d_nCols),
    d_data(0)
{
    d_data = new double[size()];
    memcpy(d_data, other.d_data, size() * sizeof(double));
}
```

```
catch (...)
{
    cerr << "Memory allocation failed\n";
    delete[] d_data;
}
```

matrix/matrix4.cc

```
#include "matrix.ih"

Matrix::Matrix(IniList iniList)
try
:
    d_nRows(iniList.size()),
    d_nCols(iniList.begin()->size()),
    d_data(0)
{
    d_data = new double[size()];
    auto ptr = d_data;
    for (auto &list: iniList)
    {
        if (list.size() != d_nCols)
            throw "Matrix(IniList): varying number of elements in rows";

        memcpy(ptr, &*list.begin() , list.size() * sizeof(double));
        ptr += list.size();
    }
}
catch (char const *message)
{
    cerr << "Exception: " << message << '\n';
    delete[] d_data;
}
catch (...)
{
    cerr << "Memory allocation failed\n";
    delete[] d_data;
}
```

matrix/operatorassign1.cc

```
// this function offers the strong guarantee. The copy construction might
// throw an excpetion, but this keeps the current data intact. Only if the
// copying succeeds, the data modified by operations which are guaranteed
// not to throw

#include "matrix.ih"

Matrix &Matrix::operator=(Matrix const &other)
{
    Matrix tmp(other);
    swap(tmp);
    return *this;
}
```

matrix/operatorassign2.cc

```
// this function offers the nothrow guarantee, because both
// of its operations offer this.

#include "matrix.ih"

Matrix &Matrix::operator=(Matrix &&tmp)
{
    swap(tmp);
    return *this;
}
```

matrix/swap.cc

```
// the swap function satisfies the nothrow guarantee, because it uses
// only primitive type operations and the C-function memcpy.
// We add the noexcept specifier to indicate that our
// code was not written to cope with a throw

#include "matrix.ih"

void Matrix::swap(Matrix &other) noexcept
{
    char buffer[sizeof(Matrix)];
    memcpy(buffer, this,   sizeof(Matrix));
    memcpy(this,   &other, sizeof(Matrix));
    memcpy(&other, buffer, sizeof(Matrix));
}
```

matrix/tr.cc

```
// This function offers the strong guarantee.
// The data is only modified if no exception is thrown by
// the preceding operations/check

#include "matrix.ih"

Matrix &Matrix::tr()
{
    double *dest = 0;               // RAII
    try
    {
        if (d_nRows != d_nCols)
            throw "Matrix::tr requires square matrix";

        dest = new double[size()];
        transpose(dest);
        delete[] d_data;
        d_data = dest;
        return *this;
    }
    catch (char const *message)
    {
        cerr << "Exception: " << message << '\n';
        throw;
    }
    catch (...)
    {
        delete[] dest;
        cerr << "Could not make transpose\n";
        throw;
    }
}
```

matrix/swap.cc

```
// the swap function satisfies the nothrow guarantee, because it uses
// only primitive type operations and the C-function memcpy.
// We add the noexcept specifier to indicate that our
// code was not written to cope with a throw

#include "matrix.ih"

void Matrix::swap(Matrix &other) noexcept
{
    char buffer[sizeof(Matrix)];
    memcpy(buffer, this,   sizeof(Matrix));
    memcpy(this,   &other, sizeof(Matrix));
    memcpy(&other, buffer, sizeof(Matrix));
}
```

matrix/transpose.cc

```
// This function offers the basic guarantee. If it cannot make a transpose,
// the allocated memory is returned.

#include "matrix.ih"

Matrix Matrix::transpose() const
{
    Matrix ret;
    try
    {
        ret.d_nCols = d_nRows;                  // prepare the return Matrix
        ret.d_nRows = d_nCols;
        ret.d_data = new double[size()];

        transpose(ret.d_data);

        return ret;
    }
    catch (...)
    {
        delete[] ret.d_data;
        cerr << "Could not make transpose\n";
        throw;
    }
}
```

Exercise 14

Exercise 15

Exercise 16

Exercise 17

Exercise 18

Exercise 19