Exercises week 1

Klaas Isaac Bijlsma s2394480
David Vroom s2309939

November 21, 2017

Exercise 1

Attain some familiarity with the way functions are selected from namespaces

We used the following code,

main.cc

```
1  #include <iostream>
2
3  namespace First
4  {
5      enum Enum
6      {};
7
8      void fun(First::Enum symbol)
9      {
10         std::cout << "First::fun called\n";
11     }
12 }
13
14 namespace Second
15 {
16     void fun(First::Enum symbol)
17     {
18         std::cout << "Second::fun called\n";
19     }
20 }
```

```
21
22 int main()
23 {
24     First::Enum symbol;
25
26     fun(symbol);             // First::fun called
27 }
```

Call fun and explain why `First::fun` is called. How would you call `Second::fun` instead?

Als een functie uit een namespace wordt aangeroepen zonder de namespace te speciferen, dan wordt de namespace van het argument van de functie gebruikt om de namespace van de functie te bepalen; het zogenaamde 'Koenig Lookup'. Aangezien het argument is gedeclareerd als type `First::Enum` wordt `First::fun` aangeroepen. Om `Second::fun` aan te roepen moet de namespace expliciet worden genoemd: `Second::fun(symbol)`.

In the namespaces slides (#6) it is stated that `operator<<`'s use is simplified because of the Koenig lookup. Explain.

Zonder Koenig lookup zal de korte versie `std::cout << "Hello"` (net als `operator<<(std::cout, "Hello")`) niet gebruikt kunnen worden. De insertion operator functie uit de standard namespace zou dan niet bereikbaar zijn zonder de prefix `std::` en de explicite functie call `operator<<(std::cout, "Hello")`.

Now, just above main, declare a function `void fun(First::Enum symbol)`. Compile this program. What happens? Why?

Er ontstaat een ambiguititeit. De compiler weet nu niet of hij de functie uit de namespace `First` of de globale functie net boven main moet aanroepen.

Exercise 2

Learn why streams can be used to determine the truth values of conditions, but not to assign values to bool variables.

Note: The code given in the exercise is incomplete, and therefore won't compile even without the intended mistake. So first of all we state the following code as a starting point:

header.ih

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6 bool promptGet(istream &in, string &str);
7 void process(string const &str);
```

main.cc

```
1 #include "header.ih"
2
3 int main()
4 {
5     string str;
6     while (promptGet(cin, str))
7         process(str);
8 }
```

process.cc

```
1 #include "header.ih"
2
3 void process(string const &str)
4 {
5     cout << "processed: " << str << '\n';
6 }
```

promptget.cc

```
1 #include "header.ih"
2
3 bool promptGet(istream &in, string &str)
4 {
5     cout << "Enter a line or ^D\n";     // ^D signals end-of-input
6
7     return getline(in, str);
8 }
```

1.

This code doesn't work, because `getline(in, str)` cannot be returned as a bool in `promptGet`. This is because the class `istream` defines `explicit operator bool() const`. This allows the compiler to only perform a conversion to a bool when this is explicitly required (as in a while statement), but not implicitly (as in the return statement above).

2.

By changing `promptGet`'s body in the following way, the code does compile:

promptget.cc

```
1 #include "header.ih"
2
3 bool promptGet(istream &in, string &str)
4 {
5     cout << "Enter a line or ^D\n";     // ^D signals end-of-input
6
7     return static_cast<bool>(getline(in, str));
8 }
```

3.

By changing `promptGet` (and the declaration in the internal header) in the following way, the code does compile:

promptget.cc

```
1 #include "header.ih"
2
3 istream &promptGet(istream &in, string &str)
4 {
5     cout << "Enter a line or ^D\n";     // ^D signals end-of-input
6
7     return getline(in, str);
8 }
```

Exercise 3

Learn to implement index operators

The Matrix class that is used here, is derived from the solutions of excercise 64. We used the following code,

matrix/matrix.h

```
#ifndef INCLUDED_MATRIX_
#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

        // exercise 5
        // ==========
    size_t d_idxColStart = 0;
    size_t d_idxRowStart = 0;
    size_t d_nColEnd = d_nCols;
    size_t d_nRowEnd = d_nRows;

    std::istream &(Matrix::*d_extractMode)(
        std::istream &in, Matrix const &matrix) const = &Matrix::extractRows;

    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
        explicit Matrix(IniList inilist);           // 4

        ~Matrix();
```

```
33
34          Matrix &operator=(Matrix const &rhs);
35          Matrix &operator=(Matrix &&tmp);
36
37          size_t nRows() const;
38          size_t nCols() const;
39          size_t size() const;            // nRows * nCols
40
41          static Matrix identity(size_t dim);
42
43          Matrix &tr();                   // transpose (must be square)
44          Matrix transpose() const;       // any dim.
45
46          void swap(Matrix &other);
47
48              // exercise 3
49              // ==========
50          double *operator[](size_t index);
51          double const *operator[](size_t index) const;
52
53              // exercise 4
54              // ==========
55          friend Matrix operator+(Matrix const &lhs, Matrix const &rhs); // 1
56          friend Matrix operator+(Matrix &&lhs, Matrix const &rhs);
    // 2
57          Matrix &operator+=(Matrix const &other) &;      // 1
58          Matrix operator+=(Matrix const &other) &&;      // 2
59
60              // exercise 5
61              // ==========
62          friend std::ostream &operator<<(
63              std::ostream &out, Matrix const &matrix);
64          friend std::istream &operator>>(
65              std::istream &in, Matrix const &matrix);
66
67          enum Mode
68          {
69              BY_ROWS,
70              BY_COLS
71          };
72
```

```
73          Matrix &operator()(
74              size_t nRows, size_t nCols, Mode byCols = BY_ROWS);     // 1
75          Matrix &operator()(
76              Mode byCols, size_t idxStart = 0, size_t nSubLines = 0); // 2
77          Matrix &operator()(
78              Mode byCols, size_t idxRowStart, size_t nSubRows,
79              size_t idxColStart, size_t nSubCols);                   // 3
80
81              // exercise 7
82              // ==========
83          friend bool operator==(Matrix const &lhs, Matrix const &rhs);
84          friend bool operator!=(Matrix const &lhs, Matrix const &rhs);
85
86      private:
87          double &el(size_t row, size_t col) const;
88          void transpose(double *dest) const;
89
90              // exercise 3
91              // ==========               // private backdoor
92          double *operatorIndex(size_t index) const;
93
94              // exercise 4
95              // ==========
96          void add(Matrix const &rhs);
97
98              // exercise 5
99              // ==========
100         std::istream &extractRows(
101             std::istream &in, Matrix const &matrix) const;
102         std::istream &extractCols(
103             std::istream &in, Matrix const &matrix) const;
104
105 };
106
107 inline size_t Matrix::nCols() const
108 {
109     return d_nCols;
110 }
111
112 inline size_t Matrix::nRows() const
113 {
```

```
114     return d_nRows;
115 }
116
117 inline size_t Matrix::size() const
118 {
119     return d_nRows * d_nCols;
120 }
121
122 inline double &Matrix::el(size_t row, size_t col) const
123 {
124     return d_data[row * d_nCols + col];
125 }
126
127     // exercise 3
128     // ==========
129 inline double *Matrix::operatorIndex(size_t index) const
130 {
131     return d_data + index * d_nCols;
132 }
133
134 inline double *Matrix::operator[](size_t index)
135 {
136     return operatorIndex(index);
137 }
138
139 inline double const *Matrix::operator[](size_t index) const
140 {
141     return operatorIndex(index);
142 }
143
144 #endif
```

Exercise 4

Learn to implement and spot opportunities for overloaded operators

The header is shown in exercise 3, the implementations of the added functions are shown below:

matrix/add.cc

```
1  #include "matrix.ih"
2
3  void Matrix::add(Matrix const &rhs)
4  {
5      if (rhs.d_nCols != d_nCols or rhs.d_nRows != d_nRows)
6      {
7          cerr << "Warning: Matrices have differnt size, "
8                  "so cannot be added!\n";
9          exit(1);
10     }
11
12     for (size_t idx = size(); idx--; )
13         d_data[idx] += rhs.d_data[idx];
14 }
```

matrix/operatoradd1.cc

```
1  #include "matrix.ih"
2
3  Matrix operator+(Matrix const &lhs, Matrix const &rhs)
4  {
5      Matrix tmp{ lhs };
6      tmp.add(rhs);
7      return tmp;
8  }
```

matrix/operatoradd2.cc

```
1  #include "matrix.ih"
2
```

```
3  Matrix operator+(Matrix &&lhs, Matrix const &rhs)
4  {
5      lhs.add(rhs);
6      return move(lhs);
7  }
```

matrix/operatorcompadd1.cc

```
1  #include "matrix.ih"
2
3  Matrix &Matrix::operator+=(Matrix const &other) &
4  {
5      Matrix tmp{ *this };
6      tmp.add(other);
7      swap(tmp);
8      return *this;
9  }
```

matrix/operatorcompadd2.cc

```
1  #include "matrix.ih"
2
3  Matrix Matrix::operator+=(Matrix const &other) &&
4  {
5      add(other);
6      return move(*this);
7  }
```

Exercise 5

Learn to insert/extract objects of your own class

We used the following code,

matrix/extractcols.cc

```
1  #include "matrix.ih"
2
3  std::istream &Matrix::extractCols(
4      std::istream &in, Matrix const &matrix) const
5  {
6      for (size_t colIdx = matrix.d_idxColStart;
7           colIdx != matrix.d_nColEnd;
8           ++colIdx)
9          for (size_t rowIdx = matrix.d_idxRowStart;
10              rowIdx != matrix.d_nRowEnd;
11              ++rowIdx)
12             in >> matrix.el(rowIdx, colIdx);
13     return in;
14 }
```

matrix/extractrows.cc

```
1  #include "matrix.ih"
2
3  std::istream &Matrix::extractRows(
4      std::istream &in, Matrix const &matrix) const
5  {
6      for (size_t rowIdx = matrix.d_idxRowStart;
7           rowIdx != matrix.d_nRowEnd;
8           ++rowIdx)
9          for (size_t colIdx = matrix.d_idxColStart;
10              colIdx != matrix.d_nColEnd;
11              ++colIdx)
12             in >> matrix.el(rowIdx, colIdx);
13     return in;
14 }
```

matrix/functor1.cc

```
#include "matrix.ih"

Matrix &Matrix::operator()(size_t nRows, size_t nCols, Mode byCols)
{
    Matrix tmp{ nRows, nCols };
    swap(tmp);
    if (byCols)
        d_extractMode = &Matrix::extractCols;
    return *this;
}
```

matrix/functor2.cc

```
#include "matrix.ih"

Matrix &Matrix::operator()(Mode byCols, size_t idxStart, size_t nSubLines)
{
    if (byCols)
    {
        d_extractMode = &Matrix::extractCols;

        if (idxStart >= d_nCols)
        {       // if requested submatrix lies outside matrix, do nothing
            d_idxColStart = d_nColEnd;
            return *this;
        }
        d_idxColStart = idxStart;
                // if number of sublines is not default and
                // submatrix lies within matrix, then set end of submatrix
        if (nSubLines == true and d_idxColStart + nSubLines < d_nCols)
            d_nColEnd = d_idxColStart + nSubLines;
    }
    else        // extract by rows
    {
        if (idxStart >= d_nRows)
        {       // if requested submatrix lies outside matrix, do nothing
            d_idxRowStart = d_nRowEnd;
            return *this;
```

```
26          }
27          d_idxRowStart = idxStart;
28                  // if number of sublines is not default and
29                  // submatrix lies within matrix, then set end of submatrix
30          if (nSubLines == true and d_idxRowStart + nSubLines < d_nRows)
31              d_nRowEnd = d_idxRowStart + nSubLines;
32      }
33
34      return *this;
35  }
```

matrix/functor3.cc

```
1   #include "matrix.ih"
2
3   Matrix &Matrix::operator()(Mode byCols,
4       size_t idxRowStart, size_t nSubRows, size_t idxColStart, size_t nSubCols)
5   {
6       if (idxRowStart >= d_nRows or idxColStart >= d_nCols)
7       {                   // if submatrix lies outside matrix then do nothing
8           d_idxRowStart = d_nRowEnd;
9           d_idxColStart = d_nColEnd;
10          return *this;
11      }
12
13      d_idxRowStart = idxRowStart; // set start values submatrix
14      d_idxColStart = idxColStart;
15
16      if (byCols)
17          d_extractMode = &Matrix::extractCols;
18
19      // if within matrix set end values of submatrix
20      if (d_idxRowStart + nSubRows < d_nRows)
21          d_nRowEnd = d_idxRowStart + nSubRows;
22
23      if (d_idxColStart + nSubCols < d_nCols)
24          d_nColEnd = d_idxColStart + nSubCols;
25
26      return *this;
27  }
```

matrix/operatorextract.cc

```
#include "matrix.ih"

istream &operator>>(istream &in, Matrix const &matrix)
{
    return static_cast<istream &>(
        (matrix.*matrix.d_extractMode)(in, matrix));
}
```

matrix/operatorinsert.cc

```
#include "matrix.ih"

ostream &operator<<(ostream &out, Matrix const &matrix)
{
    for (size_t rowIdx = 0; rowIdx != matrix.d_nRows; ++rowIdx)
    {
        for (size_t colIdx = 0; colIdx != matrix.d_nCols; ++colIdx)
            out << matrix.el(rowIdx, colIdx) << " ";
        out << '\n'; // add newline after each row
    }
    return out;
}
```

Exercise 6

Exercise 7

Learn to implement and spot opportunities for overloaded operators

1.

The following two overloaded operators are added to compare two `Matrix` objects for (in)equality:

matrix/operatorequalto.cc

```
1  #include "matrix.ih"
2
3  bool operator==(Matrix const &lhs, Matrix const &rhs)
4  {
5      if (lhs.d_nCols != rhs.d_nCols or lhs.d_nRows != rhs.d_nRows)
6          return false;
7
8      for (size_t idx = lhs.size(); idx--; )
9      {
10         if (lhs.d_data[idx] != rhs.d_data[idx])
11             return false;
12     }
13     return true;
14 }
```

matrix/operatornotequalto.cc

```
1  #include "matrix.ih"
2
3  bool operator!=(Matrix const &lhs, Matrix const &rhs)
4  {
5      if (!(lhs == rhs))
6          return true;
7
8      return false;
9  }
```

2.

We modified the following code of the `Strings` class to facilitate comparing for (in)equality,

strings/strings.h

```
#ifndef EX62_STRINGS_
#define EX62_STRINGS_

#include <iosfwd>

class Strings
{
    size_t d_size = 0;
    size_t d_capacity = 1;
    std::string **d_str;

    public:
        friend bool operator==(Strings const &lhs, Strings const &rhs);
        friend bool operator!=(Strings const &lhs, Strings const &rhs);

        Strings();
        Strings(int argc, char **argv);
        Strings(char **environLike);
        Strings(Strings const &outerStrings);   // copy constructor
        Strings(Strings &&tmp);                 // move constructor

        ~Strings();
                            // copy assignment operator
        Strings &operator=(Strings const &outerStrings);
                            // move assignment operator
        Strings &operator=(Strings &&tmp);

        void swap(Strings &other);

        size_t size() const;
        size_t capacity() const;
        std::string const &at(size_t idx) const;
        std::string &at(size_t idx);

        void add(std::string const &next);
```

```
36
37          void resize(size_t newSize);
38          void reserve(size_t newCapacity);
39
40      private:
41          std::string &safeAt(size_t idx) const;  // private backdoor
42          std::string **storageArea();
43          void destroy();
44          std::string **enlarged();
45          std::string **rawPointers(size_t nPointers);
46  };
47
48  inline size_t Strings::size() const
49  {
50      return d_size;
51  }
52
53  inline size_t Strings::capacity() const
54  {
55      return d_capacity;
56  }
57
58  inline std::string const &Strings::at(size_t idx) const
59  {
60      return safeAt(idx);
61  }
62
63  inline std::string &Strings::at(size_t idx)
64  {
65      return safeAt(idx);
66  }
67
68  #endif
```

strings/operatorequalto.cc

```
1  #include "strings.ih"
2
3  bool operator==(Strings const &lhs, Strings const &rhs)
4  {
```

```
5       if (lhs.d_size != rhs.d_size) // check size of array of Strings
6           return false;
7
8       for (size_t idx = 0; idx != lhs.d_size; ++idx)
9       {                               // compare string objects
10          if (*lhs.d_str[idx] != *rhs.d_str[idx])
11              return false;
12      }
13      return true;
14  }
```

strings/operatornotequalto.cc

```
1   #include "strings.ih"
2
3   bool operator!=(Strings const &lhs, Strings const &rhs)
4   {
5       if (!(lhs == rhs))
6           return true;
7
8       return false;
9   }
```

Exercise 8

Exercise 9

Exercise 10