Programming in C/C++
Exercises set eight: Overloading

Christiaan Steenkist
Diego Ribas Gomes
Jaime Betancor Valado
Remco Bos

November 9, 2016

The `Matrix` class

We have merged all the different matrices together to form the one true `Matrix`. Here is the header file for that class so we do not have to print it time and time again.

Listing 1: matrix.ih

```
1  #include "matrix.h"
2
3  using namespace std;
```

Listing 2: matrix.h

```
1  #ifndef MATRIX_H_
2  #define MATRIX_H_
3
4  #include <cstddef>
5  #include <initializer_list>
6  #include <iostream>
7
8  // Proxy for exercise 69.
9  enum Mode
10 {
11   BY_ROWS,
12   BY_COLUMNS
```

```
13 };
14
15 class Matrix;
16 class MatExtractProxy
17 {
18   size_t d_nRows = 0;
19   size_t d_nCols = 0;
20
21   double *d_data = 0;
22
23   size_t d_start = 0;
24   size_t d_steps = 0; // 0 means all
25
26   Mode d_mode = BY_ROWS;
27
28   public:
29     MatExtractProxy(size_t nRows, size_t nCols,
30       double *d_data, size_t start, size_t steps,
31       Mode mode);
32
33     std::istream rowExtractor(std::istream &input);
34     std::istream columnExtractor(std::istream &input);
35
36     friend std::istream &operator>>(
37       std::istream &input, MatExtractProxy &&temp);
38 };
39
40 // Matrix class.
41 class Matrix
42 {
43   size_t d_nRows = 0;
44   size_t d_nCols = 0;
45
46   double *d_data = 0;
47
48   public:
49     Matrix() = default;
50     Matrix(size_t nRows, size_t nCols);
51     Matrix(std::initializer_list<
52       std::initializer_list<double>> rows);
```

```
53      Matrix(Matrix const &other);
54      Matrix(Matrix &&other);
55      ~Matrix();
56      Matrix &operator=(Matrix const &other);
57      Matrix &operator=(Matrix &&other);
58      Matrix &tr();
59
60      void swap(Matrix &other);
61
62      size_t const &nRows();
63      size_t const &nCols();
64
65      double *row(size_t index);
66      double const *row(size_t index) const;
67
68      Matrix transpose();
69      static Matrix identity(size_t dim);
70
71      // Exercise 67
72      double *operator[](size_t rowIdx);
73      double const *operator[](size_t rowIdx) const;
74
75      // Exercise 68
76      friend Matrix operator+(Matrix const &left,
77        Matrix const &right);
78      Matrix &operator+=(Matrix const &other);
79
80      // Exercise 69
81      MatExtractProxy operator()(size_t nRows,
82        size_t nCols, Mode mode = BY_ROWS);
83      MatExtractProxy operator()(Mode mode);
84      MatExtractProxy operator()(size_t start);
85      MatExtractProxy operator()(Mode mode,
86        size_t start, size_t steps = 0);
87
88      friend std::istream &operator>>(
89        std::istream &input, Matrix &rvalue);
90      friend std::ostream &operator<<(
91        std::ostream &output, Matrix const &rvalue);
92
```

```
93      // Exercise 70
94      friend bool operator==(Matrix const &left,
95        Matrix const &right);
96      friend bool operator!=(Matrix const &left,
97        Matrix const &right);
98
99    private:
100 };
101
102 // Stream operators for exercise 69.
103 std::istream &operator>>(std::istream &input,
104   MatExtractProxy &&temp);
105 std::istream &operator>>(std::istream &input,
106   Matrix &rvalue);
107 std::ostream &operator<<(std::ostream &output,
108   Matrix const &rvalue);
109
110 // Exercise 68
111 Matrix operator+(Matrix const &left,
112   Matrix const &right);
113
114 // Exercise 70
115 bool operator==(Matrix const &left,
116   Matrix const &right);
117 bool operator!=(Matrix const &left,
118   Matrix const &right);
119
120 #endif
```

Exercise 67, the index operator

We implemented the index operator for the `Matrix` class. It returns a pointer to the first double of a row in the matrix. We also made a const variant of this function.

Code listings

Listing 3: index operator

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

```
2
3  double *Matrix::operator[](size_t rowIdx)
4  {
5      return row(rowIdx);
6  }
```

Listing 4: const index operator

```
1  #include "matrix.ih"
2
3  double const *Matrix::operator[](size_t rowIdx) const
4  {
5      return row(rowIdx);
6  }
```

Exercise 68, addition operators

We implemented the ”+” and ”+=” operators for our matrix class. Matrices can only be added if the dimensions are equal.

Code listings

Listing 5: operator+.cc

```
1  #include "matrix.ih"
2
3  Matrix operator+(Matrix const &left,
4    Matrix const &right)
5  {
6    if (left.d_nRows != right.d_nRows
7      || left.d_nCols != right.d_nCols)
8    {
9      cerr << "Matrices must have the same size.\n";
10     return left;
11   }
12
13   Matrix copy(left);
14   size_t end = copy.d_nRows * copy.d_nCols
15   for (size_t idx = 0; idx != end; ++idx)
16     copy.d_data[idx] += right.d_data[idx];
17   return copy;
18 }
```

Listing 6: operator+=.cc

```
1  #include "matrix.ih"
2
3  Matrix &Matrix::operator+=(Matrix const &other)
4  {
5    if (d_nRows != other.d_nRows
6      || d_nCols != other.d_nCols)
7    {
8      cerr << "Matrices must have the same size.";
9      return *this;
10   }
11
12   size_t end = d_nRows * d_nCols;
13   for (size_t idx = 0; idx != end; ++idx)
14     d_data[idx] += other.d_data[idx];
15   return *this;
16 }
```

Exercise 69, insertion and extraction

We implemented the insertion and extraction for the `Matrix` class. For insertion we used a single function that is used on matrices. For extraction we made a matrix function and a proxy function. The proxies are made using the overloaded parenthesis operator and the different parenthesis operators give the proxy different settings. The main function shows the slightly changed syntax compared to the code in the assignment.

Code listings

Listing 7: main.cc

```
1  #include "main.h"
2  #include <iostream>
3
4  using namespace std;
5
6  int main(int argc, char **argv)
7  {
8    Matrix mat1(4, 8);
9    cout << mat1 << '\n';
```

```
10
11    cin >> mat1;
12    cout << mat1 << '\n';
13
14    cin >> mat1(2, 5);
15    cout << mat1 << '\n';
16
17    cin >> mat1(2, 5, BY_COLUMNS);
18    cout << mat1 << '\n';
19
20    Matrix mat2(4, 8);
21    cin >> mat2(BY_COLUMNS);
22    cout << mat2 << '\n';
23
24    cin >> mat2(BY_ROWS, 2, 4);
25    cout << mat2 << '\n';
26  }
```

Parenthesis operators and Proxy

Listing 8: proxy.cc

```
1  #include "matrix.ih"
2
3  Proxy::Proxy(Matrix *matrix)
4  :
5    d_matrix(matrix)
6  {
7  }
```

Listing 9: rowextractor.cc

```
1  #include "matrix.ih"
2
3  istream &MatExtractProxy::rowExtractor(istream &input)
4  {
5    // Row-wise can directly loop through
6    // all required entries.
7    double *begin = d_data + d_start * d_nCols;
8    double *end = begin + d_steps * d_nCols;
9    for (double *place = begin; place != end; ++place)
10     input >> *place;
```

```
11
12    return input;
13  }
```

Listing 10: columnextractor.cc

```
1  #include "matrix.ih"
2
3  istream &MatExtractProxy::rowExtractor(istream &input)
4  {
5    // Column-wise needs a column step and a row step.
6    double *colStart = d_data + d_start;
7    double *colEnd = colStart + d_steps;
8    for (double *col = colStart; col != colEnd; ++col)
9    {
10     for (size_t row = 0; row != d_nRows; ++row)
11     {
12       double *place = col + row * d_nCols;
13       input >> *place;
14     }
15   }
16   return input;
17 }
```

Listing 11: poperator1.cc

```
1  #include "matrix.ih"
2
3  Proxy Matrix::operator()(size_t nRows,
4    size_t nCols, Mode mode)
5  {
6    delete[] d_data;
7
8    d_nRows = nRows;
9    d_nCols = nCols;
10   d_data = new double[nRows * nCols];
11
12   return Proxy(nRows, nCols, d_data, 0,
13     mode == BY_ROWS ? nRows : nCols, mode);
14 }
```

Listing 12: poperator2.cc

```
1  #include "matrix.ih"
2
3  Proxy Matrix::operator()(Mode mode)
4  {
5    return Proxy(d_nRows, d_nCols, d_data, 0,
6      mode == BY_ROWS ? d_nRows : d_nCols,
7      mode);
8  }
```

Listing 13: poperator3.cc

```
1  #include "matrix.ih"
2
3  Proxy Matrix::operator()(Mode mode, size_t start,
4    size_t parts)
5  {
6    return Proxy(d_nRows, d_nCols, d_data,
7      start, parts, mode);
8  }
```

Listing 14: poperator3b.cc

```
1  #include "matrix.ih"
2
3  Proxy Matrix::operator()(Mode mode, size_t start)
4  {
5    return Proxy(d_nRows, d_nCols, d_data,
6      start, 0, mode);
7  }
```

Insertion and extraction

Listing 15: inserter.cc

```
1  #include "matrix.ih"
2
3  ostream &operator<<(ostream &output,
4    Matrix const &rvalue)
5  {
6    for (size_t rowNum = 0; rowNum != rvalue.d_nRows;
7      ++rowNum)
```

```
8     {
9       double const *end = rvalue.row(rowNum)
10        + rvalue.d_nCols;
11      for (double const *place = rvalue.row(rowNum);
12           place != end; ++place)
13        output << *place << ' ';
14      output << '\n';
15    }
16    return output;
17  }
```

Listing 16: extractor.cc

```
1   #include "matrix.ih"
2
3   istream &operator>>(istream &input, Matrix &rvalue)
4   {
5     double *end = rvalue.d_data + rvalue.d_nRows
6       * rvalue.d_nCols;
7     for (double *place = rvalue.d_data; place != end;
8          ++place)
9       input >> *place;
10
11    return input;
12  }
```

Listing 17: proxyextractor.cc

```
1   #include "matrix.ih"
2
3   istream &operator>>(istream &input, Proxy &&pr)
4   {
5     size_t boundary = pr.d_mode == BY_ROWS
6       ? pr.d_nRows : pr.d_nCols;
7     if (pr.d_steps == 0)
8       pr.d_steps = boundary;
9
10    // Boundary limit for number of rows/columns.
11    // Insertion past the bounds is ignored.
12    if (pr.d_start >= boundary)
13        return input;
```

```
14
15    if (pr.d_start + pr.d_steps >= boundary)
16      pr.d_steps = boundary - pr.d_start;
17
18    return pr.d_mode == BY_ROWS
19      ? pr.columnExtractor(input)
20      : pr.rowExtractor(input, pr);
21  }
```

Exercise 70, equality and inequality operators

We overloaded the "==" and "!=" operators for both the `Strings` and the `Matrix` classes.

Code listings

`Strings`

Listing 18: strings.h

```
1  #ifndef STRINGS_H
2  #define STRINGS_H
3
4  #include <cstddef>
5  #include <string>
6
7  class Strings
8  {
9    std::string **d_str = new std::string*[1];
10   size_t d_size = 0;
11   size_t d_capacity = 1;
12
13   public:
14     Strings() = default;
15     Strings(size_t argc, char **argv);
16     Strings(char **environ);
17     Strings(Strings const &other);
18     Strings(Strings &&temp);
19     ~Strings();
20     Strings &operator=(Strings const &rvalue);
21
```

```
22      friend bool operator==(Strings const &left,
23        Strings const &right);
24      friend bool operator!=(Strings const &left,
25        Strings const &right);
26
27      void swap(Strings &other);
28
29      size_t size();
30      size_t capacity();
31
32      void resize(size_t size);
33      void reserve(size_t size);
34
35      void addString(std::string newString);
36      void addString(char *charArray);
37
38      std::string at(size_t index);
39      std::string const at(size_t index) const;
40
41      std::string **rawPointers(size_t amount);
42  };
43
44  bool operator==(Strings const &left,
45    Strings const &right);
46  bool operator!=(Strings const &left,
47    Strings const &right);
48
49  #endif
```

Listing 19: operator==.cc

```
1   #include "strings.ih"
2
3   bool operator==(Strings const &left,
4     Strings const &right)
5   {
6     if (left.d_size != right.d_size)
7       return false;
8
9     for (size_t idx = 0; idx != left.d_size; ++idx)
10      if ((*left.d_str[idx]).compare(
```

```
11          *right.d_str[idx]) != 0)
12        return false;
13
14    return true;
15  }
```

Listing 20: operator!=.cc

```
1  #include "strings.ih"
2
3  bool operator!=(Strings const &left,
4    Strings const &right)
5  {
6    return !(left == right);
7  }
```

Matrix

Listing 21: operator==.cc

```
1  #include "matrix.ih"
2
3  bool operator==(Matrix const &left,
4    Matrix const &right)
5  {
6    if (left.d_nRows != right.d_nRows
7        || left.d_nCols != right.d_nCols)
8      return false;
9
10   size_t end = left.d_nRows * left.d_nCols;
11   for (size_t index = 0; index != end;
12       ++index)
13     if (left.d_data[index] != right.d_data[index])
14       return false;
15
16   return true;
17 }
```

Listing 22: operator!=.cc

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

```
3  bool operator!=(Matrix const &left,
4    Matrix const &right)
5  {
6    return !(left == right);
7  }
```

Exercise 71, explicit constructors

What constructors are potential candidates for 'explicit'? The answer is constructors with only one parameter. They can act like implicit converters, so to avoid this we should use the keyword "explicit" to allow only explicit conversions.

Code listing

Listing 23: strings.h

```
1  #ifndef STRINGS_H
2  #define STRINGS_H
3
4  #include <cstddef>
5  #include <string>
6
7  class Strings
8  {
9    std::string **d_str = new std::string*[1];
10   size_t d_size = 0;
11   size_t d_capacity = 1;
12
13   public:
14     Strings() = default;
15     // Modified constructors
16     // with explicit keyword
17     explicit Strings(size_t argc, char **argv);
18     explicit Strings(char **environ);
19
20     Strings(Strings const &other);
21     Strings(Strings &&temp);
22     ~Strings();
23     Strings &operator=(Strings const &rvalue);
24
25     void swap(Strings &other);
```

```
26
27      size_t size();
28      size_t capacity();
29
30      void resize(size_t size);
31      void reserve(size_t size);
32
33      void addString(std::string newString);
34      void addString(char *charArray);
35
36      std::string at(size_t index);
37      std::string const at(size_t index) const;
38
39      std::string **rawPointers(size_t amount);
40  };
41
42  #endif
```