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

Christiaan Steenkist
Diego Ribas Gomes
Jaime Betancor Valado
Remco Bos

November 3, 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 Proxy
17  {
18    Matrix *d_matrix;
19
20    size_t d_start = 0;
21    size_t d_steps = 0; // 0 means all
22
23    Mode d_mode = BY_ROWS;
24
25    public:
26      Proxy(Matrix *matrix, size_t start, size_t steps,
27        Mode mode);
28      friend std::istream &operator>>(
29        std::istream &input, Proxy &&temp);
30  };
31
32  // Matrix class.
33  class Matrix
34  {
35    size_t d_nRows = 0;
36    size_t d_nCols = 0;
37
38    double *d_data = 0;
39
40    public:
41      Matrix() = default;
42      Matrix(size_t nRows, size_t nCols);
43      Matrix(std::initializer_list<
44        std::initializer_list<double>> rows);
45      Matrix(Matrix const &other);
46      Matrix(Matrix &&other);
47      ~Matrix();
48      Matrix &operator=(Matrix const &other);
49      Matrix &operator=(Matrix &&other);
50      Matrix &tr();
51
52      void swap(Matrix &other);
```

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

```
93    Proxy &&temp);
94  std::istream &operator>>(std::istream &input,
95    Matrix &rvalue);
96  std::ostream &operator<<(std::ostream &output,
97    Matrix const &rvalue);
98
99  #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

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 3: 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    Matrix copy(other);
13    for (size_t idx = 0; idx != d_nRows * d_nCols;
14      ++idx)
```

```
15      copy.d_data[idx] += d_data[idx];
16    return copy;
17  }
```

Listing 4: 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   for (size_t idx = 0; idx != d_nRows * d_nCols;
13     ++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 5: 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 6: proxy.cc

```
1  #include "matrix.ih"
2
3  Proxy::Proxy(Matrix *matrix, size_t start,
4    size_t steps, Mode mode)
5  :
6    d_matrix(matrix),
7    d_start(start),
8    d_steps(steps),
9    d_mode(mode)
10 {
11 }
```

Listing 7: poperator1.cc

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

```
3  Proxy Matrix::operator()(size_t nRows, size_t nCols,
4    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(this, 0,
13     mode == BY_ROWS ? nRows : nCols, mode);
14 }
```

Listing 8: poperator2.cc

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

Listing 9: poperator3.cc

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

Listing 10: poperator3b.cc

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

Insertion and extraction

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

Listing 12: extractor.cc

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

Listing 13: proxyextractor.cc

```
1  #include "matrix.ih"
2
3  istream &operator>>(istream &input, Proxy &&pr)
```

```
4  {
5    size_t const nRows = pr.d_matrix->nRows();
6    size_t const nCols = pr.d_matrix->nCols();
7    size_t boundary = pr.d_mode == BY_ROWS
8      ? nRows : nCols;
9    if (pr.d_steps == 0)
10     pr.d_steps = boundary;
11
12   // Boundary limit for number of rows/columns.
13   // Insertion past the bounds is ignored.
14   if (pr.d_start >= boundary)
15       return input;
16
17   if (pr.d_start + pr.d_steps >= boundary)
18     pr.d_steps = boundary - pr.d_start;
19
20   if (pr.d_mode == BY_ROWS)
21   {
22     // Row-wise can directly loop through
23     // all required entries.
24     double *begin = pr.d_matrix->row(pr.d_start);
25     double *end = begin + pr.d_steps * nCols;
26     for (double *place = begin; place != end; ++place)
27       input >> *place;
28   }
29   else
30   {
31     // Column-wise needs a column step and a row step.
32     double *colStart = pr.d_matrix->row(0)
33       + pr.d_start;
34     double *colEnd = colStart + pr.d_steps;
35     for (double *col = colStart; col != colEnd; ++col)
36     {
37       for (size_t row = 0; row != nRows; ++row)
38       {
39         double *place = col + row * nCols;
40         input >> *place;
41       }
42     }
43   }
```

```
44
45    return input;
46  }
```

Exercise 70, equality and inequality operators

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

Code listings

`Strings`

Listing 14: 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     bool operator==(Strings const &rvalue);
23     bool 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
```

Listing 15: operator==.cc

```
1  #include "strings.ih"
2
3  bool Strings::operator==(Strings const &rvalue)
4  {
5    if (d_size != rvalue.d_size)
6      return false;
7
8    for (size_t index = 0; index != d_size; ++index)
9    {
10     string lstring = *d_str[index];
11     string rstring = *rvalue.d_str[index];
12     if (lstring.compare(rstring) != 0)
13       return false;
14   }
15
16   return true;
17 }
```

Listing 16: operator!=.cc

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

```
3  bool Strings::operator!=(Strings const &rvalue)
4  {
5    return !(*this == rvalue);
6  }
```

Matrix

Listing 17: operator==.cc

```
1  #include "matrix.ih"
2
3  bool Matrix::operator==(Matrix const &rvalue)
4  {
5    if (d_nRows != rvalue.d_nRows)
6      return false;
7    if (d_nCols != rvalue.d_nCols)
8      return false;
9
10   for (size_t index = 0; index != d_nRows * d_nCols;
11     ++index)
12   {
13     if (d_data[index] != rvalue.d_data[index])
14       return false;
15   }
16
17   return true;
18 }
```

Listing 18: operator!=.cc

```
1  #include "matrix.ih"
2
3  bool Matrix::operator!=(Matrix const &rvalue)
4  {
5    return !(*this == rvalue);
6  }
```

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 19: strings.h

```
#ifndef STRINGS_H
#define STRINGS_H

#include <cstddef>
#include <string>

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

  public:
    Strings() = default;
    // Modified constructors
    // with explicit keyword
    explicit Strings(size_t argc, char **argv);
    explicit Strings(char **environ);

    Strings(Strings const &other);
    Strings(Strings &&temp);
    ~Strings();
    Strings &operator=(Strings const &rvalue);

    void swap(Strings &other);

    size_t size();
    size_t capacity();

    void resize(size_t size);
    void reserve(size_t size);

    void addString(std::string newString);
    void addString(char *charArray);

    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
```