Programming in C/C++
Exercises set seven: Allocation in Classes

Christiaan Steenkist
Diego Ribas Gomes
Jaime Betancor Valado
Remco Bos

November 2, 2016

Assignment 58, `Strings` as a value-class

We implemented the copy and move constructor as well as the assignment operator.

Code listings

Listing 1: strings.h

```
#ifndef STRINGS_H
#define STRINGS_H

#include <string>

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

  public:
    Strings() = default;
    Strings(size_t argc, char **argv);
    Strings(char **environ);
    Strings(Strings const &other);
    Strings(Strings &&temp);
    ~Strings();
    Strings &operator=(Strings const &rvalue);
    Strings &operator=(Strings &&temp);

    void swap(Strings &other);

    std::size_t size();
    std::size_t capacity();

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

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

    std::string at(std::size_t index);
    std::string const at(std::size_t index) const;

  private:
    std::string **rawPointers(std::size_t amount);
};

#endif
```

Listing 2: Copy constructor

```
#include "strings.ih"

Strings::Strings(Strings const &other)
:
  d_str(new string *[other.d_capacity]),
  d_size(other.d_size),
  d_capacity(other.d_capacity)
{
  for (size_t index = 0; index != d_size; ++index)
    d_str[index] = new string(*other.d_str[index]);
}
```

Listing 3: Move constructor

```
#include "strings.ih"
```

```

Strings::Strings(Strings &&temp)
:
  d_str(temp.d_str),
  d_size(temp.d_size),
  d_capacity(temp.d_capacity)
{
  temp.d_str = 0;
  temp.d_size = 0;
  temp.d_capacity = 0;
}
```

Listing 4: Assignment operator

```
#include "strings.ih"

Strings &Strings::operator=(Strings const &rvalue)
{
  return *this = Strings(rvalue);
}
```

Listing 5: Move assignment operator

```
#include "strings.ih"

Strings &Strings::operator=(Strings &&temp)
{
  swap(temp);
  return *this;
}
```

Listing 6: swap.cc

```
#include "strings.ih"

void Strings::swap(Strings &other)
{
  char bytes[sizeof(Strings)];
  memcpy(bytes, this, sizeof(Strings));
  memcpy(this, &other, sizeof(Strings));
  memcpy(&other, bytes, sizeof(Strings));
}
```

Assignment 59, swapping

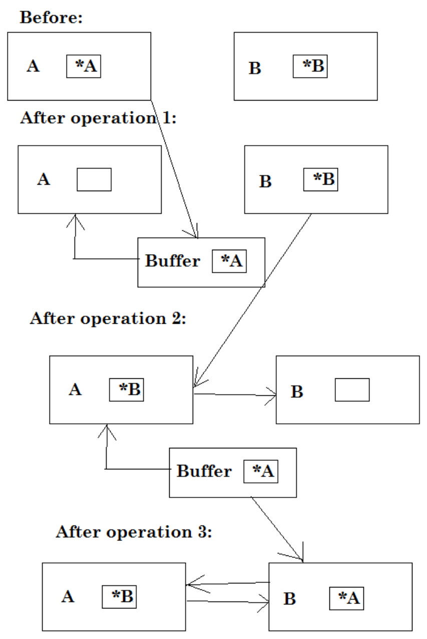
Here we answer some questions about swapping.

Example code

```cpp
void Class::swap(Class &other)
{
  char buffer[sizeof(Class)];
  memcpy(buffer,  this,   sizeof(Class)); //Operation 1
  memcpy(this, &other, sizeof(Class));  //Operation 2
  memcpy(&other, buffer,  sizeof(Class)); //Operation 3
}
```

Explanation

Figure 1: A before-after diagram



Before the fast swap is done, an object A has a pointer data member *A to itself and an object B has a pointer data member *B to itself. Pointers are variables that hold a memory address (in this case its current object). The pointers keep pointing to the memory address they were defined for. After operation 1, the pointer *A is moved to the buffer, but still points to its previous current object A. After operation 2, the pointer *B is moved to object A, but still points to its previous current object B. After operation 3, the pointer *A is moved to object B, and is still pointing to object A. This results in a situation where the objects have swapped data members, however they now point to each other. So the self-pointing property of the class is gone.

Using the keyword ’this’ prevents this situation. The ’this’ keyword is placed in front of the data member and then points to the current object where the data member resides. After doing the swapping like above but with the ’this’ keyword, the data members of A and B are swapped. They still have the ’this’ keyword in front of them and now it still a self-pointing class.

Assignment 60, the default keyword

In the example of slide 30, ’default’ wasn’t specified for the copy constructor because it would defeat the purpose of explicitly defining it, which is to prevent wild pointers. In that case, the trivial copy constructor would be used, copying struct- or class- objects member-wise and, considering that the class allocates memmory for its own use, resulting in wild pointers.

One case where specifying default for a copy constructor could make sense would be for a class that simultaneously:

1. does NOT have pointer data members pointing at its own data and
2. have had its copy constructor suppressed by declarations of either the move constructor or the move assignment operator.

Another would be to default the copy constructor in the header file as a form of documentation, to make it clear for the users.

Assignment 61, copy elision

Here we show some situations where copy elision is and isn’t used. First we are asked to define a class Demo:

Listing 7: Demo.h

```
#ifndef DEMO_H
```

```cpp
#define DEMO_H

#include <iostream>
#include <string>

class Demo
{
  int **d_entry;
  size_t d_size;

  public:
    Demo()  //Constructor
    :
      d_entry(new int*[1]),
      d_size(0)
    {
      std::cout << "This is a default constructor"
            << std::endl;
    }
    Demo(Demo const &other) //Copy constructor
    :
      d_entry(new int*[other.d_size]),
      d_size(other.d_size)
    {
      std::cout << "This is a copy constructor"
            << std::endl;

      for (size_t index = 0; index < d_size; ++index)
        d_entry[index] =
          new int(*other.d_entry[index]);
    }
    Demo &operator=(Demo const &rvalue) //CAO
    {
      std::cout << "This is a copy assignment"
            " operator" << std::endl;

      delete[] d_entry;

      d_size = rvalue.d_size;
      d_entry = new int*[d_size];

      for (size_t index = 0; index < d_size; ++index)
        d_entry[index] =
          new int(*rvalue.d_entry[index]);

      return *this;
    }
    ~Demo() //Destructor
    {
      std::cout << "This is a destructor"
            << std::endl;
      for (size_t index = 0; index < d_size; ++index)
        delete d_entry[index];
      delete[] d_entry;
    }
    Demo(Demo &&tmp)  //move constructor
    :
      d_entry(tmp.d_entry),
      d_size(tmp.d_size)
    {
      std::cout << "This is a move constructor"
            << std::endl;

      tmp.d_entry = 0;
      tmp.d_size = 0;
    }
    Demo &operator=(Demo &&other) //MAO
    {
      std::cout << "This is a move assignment"
            " operator" << std::endl;

      delete[] d_entry;

      d_entry = other.d_entry;
      d_size = other.d_size;

      other.d_entry = 0;
      other.d_size = 0;

      return *this;
    }
};

#endif
```

Also, we are asked to implement a Demo function in which:

Copy elision is used

Listing 8: Copy Elision.cc

```
#include "Demo.h"

Demo factory()
{
  Demo example;
  return example;
}

int main()
{
  factory();
}
```

Move constructor is used

Listing 9: Move constructor.cc

```
#include "Demo.h"

Demo factory(Demo example)
{
  return example;
}

int main()
{
  Demo val = factory(Demo());
}
```

Copy assignment is used

Listing 10: Copy assignment.cc

```
#include "Demo.h"

int main()
{
  Demo val1, val2;
  val1 = val2;
}
```

Move assignment is used

Listing 11: Move assignment.cc

```
#include "Demo.h"

Demo factory(Demo example)
{
  return example;
}

int main()
{
  Demo val;
  val = factory(Demo());
}
```

Assignment 62

We have constructed a `Matrix` class that can handle any size matrix according to the specifications of the assignment. All requested functions have been implemented and an example main function has been made showing off valid and invalid uses.

Matrix class

Listing 12: matrix.ih

```
#include "matrix.h"
#include <cstddef>
#include <cstring>
#include <initializer_list>
#include <iostream>
```

```

using namespace std;
```

Listing 13: matrix.h

```
#ifndef MATRIX_H_
#define MATRIX_H_

#include <cstddef>
#include <initializer_list>

class Matrix
{
  size_t d_nRows = 0;
  size_t d_nCols = 0;

  double *d_data = 0;

  public:
    Matrix() = default;
    Matrix(size_t nRows, size_t nCols);
    Matrix(std::initializer_list<
      std::initializer_list<double>> rows);
    Matrix(Matrix const &other);
    Matrix(Matrix &&other);
    ~Matrix();
    Matrix &operator=(Matrix const &other);
    Matrix &operator=(Matrix &&other);
    Matrix &tr();

    void swap(Matrix &other);

    size_t const &nRows();
    size_t const &nCols();

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

    Matrix transpose();
    static Matrix identity(size_t dim);

  private:
};

#endif
```

Listing 14: copyconstructor.cc

```
#include "matrix.ih"

Matrix::Matrix(Matrix const &other)
:
  d_nRows(other.d_nRows),
  d_nCols(other.d_nCols),
  d_data(new double[other.d_nRows * other.d_nCols])
{
  size_t size = other.d_nRows * other.d_nCols;
  for (size_t index = 0; index != size; ++index)
    d_data[index] = other.d_data[index];
}
```

Listing 15: destructor.cc

```
#include "matrix.ih"

Matrix::~Matrix()
{
  delete[] d_data;
}
```

Listing 16: identity.cc

```
#include "matrix.ih"

Matrix Matrix::identity(size_t dim)
{
  Matrix identity(dim, dim);

  for (size_t index = 0; index != dim; ++index)
    *(identity.row(index) + index) = 1;

  return identity;
}
```

Listing 17: matrix1.cc

```
#include "matrix.ih"

Matrix::Matrix(size_t nRows, size_t nCols)
:
  d_nRows(nRows),
  d_nCols(nCols),
  d_data(new double[nRows * nCols])
{
  size_t size = nRows * nCols;
  for (size_t index = 0; index != size; ++index)
    d_data[index] = 0;
}
```

Listing 18: matrix2.cc

```
#include "matrix.ih"

Matrix::Matrix(initializer_list<initializer_list
  <double>> rows)
:
  d_nRows(rows.size()),
  d_nCols(rows.begin()[0].size()),
  d_data(new double[rows.size()
    * rows.begin()[0].size()])
{
  // Filling in the array with
  // the values from the list.
  size_t index = 0;
  for (initializer_list<double> cols : rows)
  {
    if (cols.size() != d_nCols)
    {
      cerr << "Matrix could not be made: "
        << "unequal amount of columns ("
        << d_nCols << " and " << cols.size()
        << ").\n";
      exit(1);
    }

    for (double value : cols)
    {
      d_data[index] = value;
      ++index;
    }
  }
}
```

Listing 19: moveconstructor.cc

```
#include "matrix.ih"

Matrix::Matrix(Matrix &&temp)
:
  d_nRows(temp.d_nRows),
  d_nCols(temp.d_nCols),
  d_data(temp.d_data)
{
  temp.d_nRows = 0;
  temp.d_nCols = 0;
  temp.d_data = 0;
}
```

Listing 20: moveoperator

```
#include "matrix.ih"

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

Listing 21: ncols.cc

```
#include "matrix.ih"

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

Listing 22: nrows.cc

```
#include "matrix.ih"

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

Listing 23: operator

```
#include "matrix.ih"

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

Listing 24: row.cc

```
#include "matrix.ih"

double *Matrix::row(size_t index)
{
  return d_data + index * d_nCols;
}
```

Listing 25: rowconst.cc

```
#include "matrix.ih"

double const *Matrix::row(size_t index) const
{
  return d_data + index * d_nCols;
}
```

Listing 26: swap.cc

```
#include "matrix.ih"

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

Listing 27: tr.cc

```
#include "matrix.ih"

Matrix &Matrix::tr()
{
  if (d_nRows != d_nCols)
  {
    cerr << "Matrix could not be transposed: "
      << "matrix not square ("
      << d_nRows << "x" << d_nCols << ").\n";
    exit(1);
  }

  // The elements of each row are swapped with
  // the elements of the corresponding column
  // up to the diagonal.
  // The diagonal remains unchanged.
  for (size_t nRow = 0; nRow != d_nRows; ++nRow)
  {
    for (size_t nCol = 0; nCol != nRow; ++nCol)
    {
      double *targetPtr = row(nRow) + nCol;
      double *mirrorPtr = row(nCol) + nRow;

      double h = *targetPtr;
      *targetPtr = *mirrorPtr;
      *mirrorPtr = h;
    }
  }

  return *this;
}
```

Main

Listing 28: main.ih

```
#include "main.h"
#include <iostream>

using namespace std;
```

Listing 29: main.h

```
#ifndef MAIN_H_
#define MAIN_H_

#include "matrix/matrix.h"

void printMatrix(Matrix matrix);

#endif
```

Listing 30: main.cc

```
#include "main.h"

int main(int argc, char **argv)
{
  Matrix matrix1(4, 7);
  Matrix matrix2 = Matrix::identity(5);
  Matrix matrix3({{1, 2, 3}, {4, 5, 6}, {7, 8, 9}});
  Matrix matrix4(matrix3.transpose());

  // Causes an error, column too big/small.
  //Matrix matrix5({{1, 2, 3}, {4, 5, 6}, {7, 8}});

  // Causes an error, matrix not square.
  //Matrix matrix6(matrix1.transpose();

  printMatrix(matrix1);
  printMatrix(matrix2);
  printMatrix(matrix3);
  printMatrix(matrix4);
}
```

Listing 31: printmatrix.cc

```cpp
#include "main.ih"

using namespace std;

void printMatrix(Matrix matrix)
{
  for (size_t nRow = 0; nRow != matrix.nRows(); ++nRow)
  {
    for (size_t nCol = 0; nCol != matrix.nCols();
      ++nCol)
    {
      cout << *(matrix.row(nRow) + nCol) << ' ';
    }
    cout << '\n';
  }
  cout << '\n';
}
```