CS 3460

Constructors & Destructors

Constructors Overview

- Constructors have the same concept and (more or less) syntax as Java
- But there is a lot more complexity in C++
- Nine topics, all related to constructors
  - default constructors (already discussed)
  - overloaded constructors (already discussed)
  - constructor delegation (somewhat discussed)
  - copy constructors
  - destructors
  - move constructors
  - inheriting constructors (already discussed)
  - r-values and r-value references
  - move assignment

A Matrix Class

- We'll use a Matrix class to demonstrate all of this

```
class Matrix
{
  public:
    Matrix();
    Matrix(std::size_t cols, std::size_t rows);
    Matrix(std::initializer_list<std::initializer_list<std::int32_t>> list);
    Matrix(const Matrix& matrix);
    Matrix(Matrix&& matrix);
    ~Matrix();

    Matrix& Matrix::operator=(Matrix&& rhs);
    std::int32_t& operator()(std::size_t row, std::size_t col);

    std::size_t getColumns() const { return m_cols; }
    std::size_t getRows() const { return m_rows; }

  private:
    std::size_t m_rows;
    std::size_t m_cols;
    std::int32_t** m_data;

    void buildMemory(std::size_t rows, std::size_t cols);
};
```

Default Constructors

- Same as Java…
  - Has no parameters
  - If no user defined default constructor, compiler writes one, unless any other user defined constructor is written

```
class Matrix
{
  public:
    Matrix();
    Matrix(std::size_t cols, std::size_t rows);
    Matrix(std::initializer_list<std::initializer_list<std::int32_t>> list);
    Matrix(const Matrix& matrix);
    Matrix(Matrix&& matrix);
};
```

Overloaded Constructors

- Same as Java, any constructor with parameters
- Note the third constructor
  - accepts an (nested) `std::initializer_list`

```
class Matrix
{
  public:
    Matrix();
    Matrix(std::size_t cols, std::size_t rows);
    Matrix(std::initializer_list<std::initializer_list<std::int32_t>> list);
    Matrix(const Matrix& matrix);
    Matrix(Matrix&& matrix);
};
```

Constructor – Overloaded

- First overloaded accepts a number of rows and columns to size the matrix
  - Using raw pointers to demonstrate destructors
  - Prefer `std::vector` otherwise

```
Matrix::Matrix(std::size_t rows, std::size_t cols)
{
    buildMemory(rows, cols);
}
```

```
void Matrix::buildMemory(std::size_t rows, std::size_t cols)
{
    m_rows = rows;
    m_cols = cols;
    std::int32_t** data = new std::int32_t*[rows];
    for (decltype(rows) row = 0; row < rows; row++)
    {
        data[row] = new std::int32_t[cols];
        std::memset(data[row], 0, sizeof(int32_t) * cols);
    }

    m_data = data;
}
```

Constructor Delegation

- Remember *constructor chaining* from Java?
  - Ability to specify which super constructor is chained
  - Uses the `super` keyword
- Have the same capability in C++
  - It is called *constructor delegation*
  - Syntax is different, but effect is the same

```
Matrix::Matrix() :
    Matrix(2, 2)
{
}
```

Constructor – `std::initializer_list`

- Begins by delegating to another overloaded constructor
- Then iterates over the `std::initializer_list`, taking values from it and placing them into internal storage

```
Matrix::Matrix(std::initializer_list<std::initializer_list<std::int32_t>> list) :
    Matrix(list.size(), list.begin()->size())
{
    std::size_t r = 0;
    for (auto row = list.begin(); row != list.end(); row++, r++)
    {
        std::size_t c = 0;
        for (auto column = row->begin(); column != row->end(); column++, c++)
        {
            m_data[r][c] = *column;
        }
    }
}
```

```
Matrix m({{0, 1, 2},
          {3, 4, 5},
          {6, 7, 8}});
```

Constructor – `std::initializer_list`

- Could we take it by reference rather than value?

```
Matrix::Matrix(std::initializer_list<std::initializer_list<std::int32_t>>& list) :
    Matrix(list.size(), list.begin()->size())
{
    std::size_t r = 0;
    for (auto row = list.begin(); row != list.end(); row++, r++)
    {
        std::size_t c = 0;
        for (auto column = row->begin(); column != row->end(); column++, c++)
        {
            m_data[r][c] = *column;
        }
    }
}
```

```
Matrix m({{0, 1, 2},
          {3, 4, 5},
          {6, 7, 8}});
```

Constructor – `std::initializer_list`

- Could we take it by reference rather than value?
  - No

This cannot be taken as a reference

```
Matrix::Matrix(std::initializer_list<std::initializer_list<std::int32_t>>& list) :
    Matrix(list.size(), list.begin()->size())
{
    std::size_t r = 0;
    for (auto row = list.begin(); row != list.end(); row++, r++)
    {
        std::size_t c = 0;
        for (auto column = row->begin(); column != row->end(); column++, c++)
        {
            m_data[r][c] = *column;
        }
    }
}
```

```
Matrix m({{0, 1, 2},
          {3, 4, 5},
          {6, 7, 8}});
```

Constructor – `std::initializer_list`

- Could we take it by reference rather than value?
  - No
  - But can take it as an r-value reference (coming soon)

```
Matrix::Matrix(std::initializer_list<std::initializer_list<std::int32_t>>&& list) :
    Matrix(list.size(), list.begin()->size())
{
    std::size_t r = 0;
    for (auto row = list.begin(); row != list.end(); row++, r++)
    {
        std::size_t c = 0;
        for (auto column = row->begin(); column != row->end(); column++, c++)
        {
            m_data[r][c] = *column;
        }
    }
}
```

```
Matrix m({{0, 1, 2},
          {3, 4, 5},
          {6, 7, 8}});
```

Copy Constructors

- No similar concept in Java
- Invoked whenever a copy of an object is made
  - Usually done when compiler needs to make a copy
  - Can be used to manually copy, but rarely done

```
class Matrix
{
  public:
    Matrix();
    Matrix(std::size_t cols, std::size_t rows);
    Matrix(std::initializer_list<std::initializer_list<std::int32_t>> list);
    Matrix(const Matrix& matrix);
    Matrix(Matrix&& matrix);

};
```

Copy Constructor – When Invoked?

- When `m2` is declared, it is initialized with the data from `m1`; copy constructor
- When `m1` is passed to `invertMatrix`; copy constructor

```
Matrix invertMatrix(Matrix m);

int main()
{
     Matrix m1({{0, 1, 2},
                {3, 4, 5},
                {6, 7, 8}});
     Matrix m2 = m1;    // copy

     Matrix m3 = invertMatrix(m1);    // copy

     return 0;
}
```

Copy Constructors – General Form

- Has a specific general form

  `[class](const [class]& obj);`

- Preferably `const` reference, but `const` is not required
- If not provided by user, compiler provides one
  - Member-by-member copy

```
class Matrix
{
  public:
    Matrix();
    Matrix(std::size_t cols, std::size_t rows);
    Matrix(initializer_list<initializer_list<int32_t>> list);
    Matrix(const Matrix& matrix);
    Matrix(Matrix&& matrix);

};
```

Copy Constructors - Implementation

- Copy values from the parameter into the object
- The object on which the constructor is being called is the new object, destination for copies of the parameter
- Semantics of the *copy* are up to the programmer

```
Matrix::Matrix(const Matrix& matrix)
{
    buildMemory(matrix.m_rows, matrix.m_cols);

    for (std::size_t row = 0; row < m_rows; row++)
    {
        std::memcpy(m_data[row], matrix.m_data[row], sizeof(int32_t) * m_cols);
    }
}
```

Destructors

- Special method automatically invoked when an object goes out of scope; complement to a constructor
- Java does not have this concept
  - Uses the `Dispose` interface (similar, but different)
- Name: `~[class name]() { … }`
  - e.g., `~Circle() { … }`
- Why destructors?
  - Release acquired resources (memory, files, etc)

Destructors

- Let's go back to the `Matrix` class
- In the constructors dynamic memory, using raw pointers, was acquired. Need to release it...in the destructor!

```
class Matrix
{
  public:
    … various constructors here …
    ~Matrix();
};
```

```
Matrix::~Matrix()
{
    cleanupMemory();
}
```

```
Matrix::cleanupMemory()
{
    if (m_data != nullptr)
    {
        for (decltype(m_rows) row = 0; row < m_rows; row++)
        {
            delete[] m_data[row];
        }
        delete[] m_data;
        m_data = nullptr;
    }
}
```

R-Value References – l-values

- Let's start with explaining an *l-value*
  - Sometimes known as a "locator value"
  - Can appear on left or right side of an expression

```
int x = 44;
int y = 66;
y = x;
```

  - In all three statements `x` and `y` are l-values, even `y` in the third statement
  - *An l-value is an object that lives beyond the expression in which it is used*

R-Value References – r-values

- An r-value is an expression that can only appear on the right-hand side of the assignment operator

```
int x = 44;
int y = 66;
int z = x * y;
```

- In the last statement, `x * y` is an *r-value*
  - Doesn't make any sense to appear on the left
- *An r-value is a temporary object that **does not** live beyond the expression in which it is used*

R-Value References

- We now understand l-values and r-values, what is an *r-value reference*?
- Let's start with an *l-value reference*

```
int x = 44;
int& xRef = x;
```

R-Value References

- We now understand l-values and r-values, what is an *r-value reference*?
- Let's start with an *l-value reference*

```
int x = 44;
int& xRef = x;
```

- Interestingly, can obtain reference to an r-value!

```
int x = 44;
int y = 66;
int&& xyRef = x * y;

std::cout << xyRef << std::endl;
```

- r-value references are declared using the `&&` decorator
- An r-value reference causes an r-value to continue to live

R-Value References – So What?

R-Value References – So What?

- Let’s start by adding the assignment operator

```
Matrix& Matrix::operator=(const Matrix& rhs)
{
    cleanupMemory();
    buildMemory(rhs.m_rows, rhs.m_cols);

    for (std::size_t row = 0; row < m_rows; row++)
    {
        std::memcpy(m_data[row], rhs.m_data[row], sizeof(int32_t) * m_cols);
    }

    return *this;
}
```

R-Value References – So What?

- Next, overload the `()` operator

```
std::int32_t& Matrix::operator()(std::size_t row, std::size_t col)
{
    return m_data[row][col];
}
```

R-Value References – So What?

- Now, let's make a copy of a matrix

```
Matrix m1(4,4);
Matrix m2;
m2 = copyMatrix(m1);
```

```
Matrix copyMatrix(Matrix m)
{
    return m;
}
```

R-Value References – So What?
Move Operations!

Move Constructors

- Move operation
  - Ownership of values from one object are transferred (or swapped) to another object
    - Transfer ownership from source to destination
  - Most often used to change ownership of pointers
  - Useful when the source object is temporary and/or is about to be destroyed

Move Constructors – General Form

- Has a specific general form

```
[class]([class]&& obj);
```

- Notice no use of `const`, because we need to modify the object passed in through the parameter
- No compiler provided default

```
class Matrix
{
  public:
    Matrix();
    Matrix(std::size_t cols, std::size_t rows);
    Matrix(std::initializer_list<std::initializer_list<std::int32_t>> list);
    Matrix(const Matrix& matrix);
    Matrix(Matrix&& matrix);

};
```

Move Constructors – Implementation

- We transfer ownership of the pointer
  - When source goes out of scope, its destructor won't try to delete the pointer

```
Matrix::Matrix(Matrix&& matrix)
{
    m_rows = matrix.m_rows;
    m_cols = matrix.m_cols;
    m_data = matrix.m_data;

    matrix.m_rows = 0;
    matrix.m_cols = 0;
    matrix.m_data = nullptr;
}
```

Move Assignment Operator

- Occurs when a temporary object is being assigned to another object
- Remember this…

```
Matrix m1;
m1 = copyMatrix(4, 4);
```

  - Assignment operator invoked using a temporary object
- Can improve by providing a move assignment operator
  - Instead of move only, swap ownership

```
Matrix& Matrix::operator=(Matrix&& rhs)
{
    if (this != &rhs)
    {
        std::swap(m_rows, rhs.m_rows);
        std::swap(m_cols, rhs.m_cols);
        std::swap(m_data, rhs.m_data);
    }

    return *this;
}
```

R-Value References & Range-Based Loops

- Consider this code

```
std::vector<std::string> cities =
    { "Paradise", "Hyrum", "Nibley", "Hyde Park", "Smithfield", "Newton" };

for (auto city : cities)
{
    std::cout << city << std::endl;
}
```

- A copy is made into `city` for each element of `cities`
- Can improve with reference or r-value reference

```
for (auto&& city : cities)     // for (auto& city : cities)
{
    std::cout << city << std::endl;
}
```

- Best when a function returning temporary objects

Rule of Five (Rule of Four?)

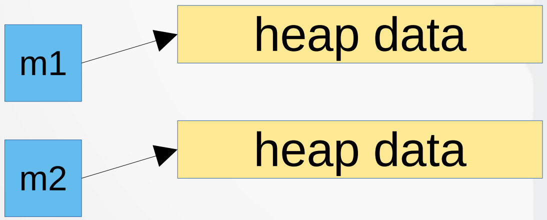
- If any one of the following are provided by the programmer, then all should be provided…
  - Copy Constructor
  - Move Constructor
  - Assignment Operator
  - Move Assignment Operator
  - Destructor (depends on nature of the class)

Move/Copy Constructors/Assignment

```
Matrix m1(2,2);
Matrix m2 = Matrix(4,4);

m2 = copyMatrix(m1);
```

```
Matrix copyMatrix(Matrix m)
{
    return m;
}
```



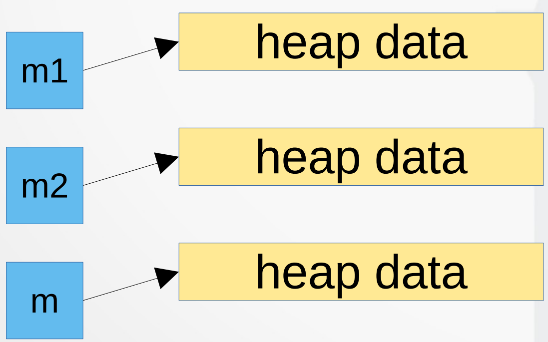
Move/Copy Constructors/Assignment

```
Matrix m1(2,2);
Matrix m2 = Matrix(4,4);

m2 = copyMatrix(m1);
```

```
Matrix copyMatrix(Matrix m)
{
    return m;
}
```

Step 1: m1 is copied into m



Move/Copy Constructors/Assignment

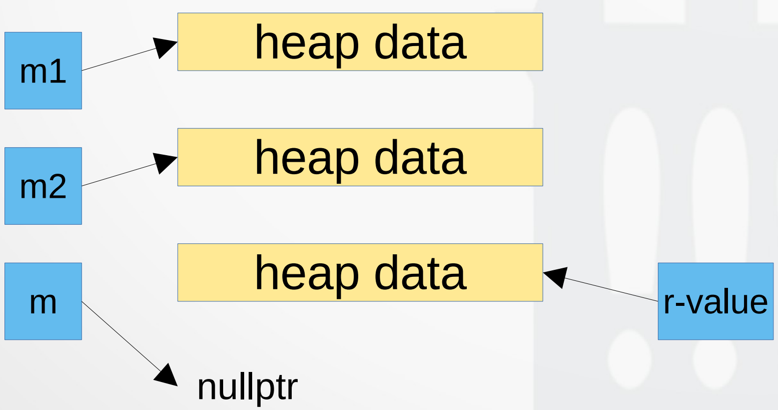
```
Matrix m1(2,2);
Matrix m2 = Matrix(4,4);

m2 = copyMatrix(m1);
```

```
Matrix copyMatrix(Matrix m)
{
    return m;  r-value
}
```

Step 1: m1 is copied into m

Step 2: m is move copied into return r-value



Move/Copy Constructors/Assignment

```
Matrix m1(2,2);
Matrix m2 = Matrix(4,4);

m2 = copyMatrix(m1);
```

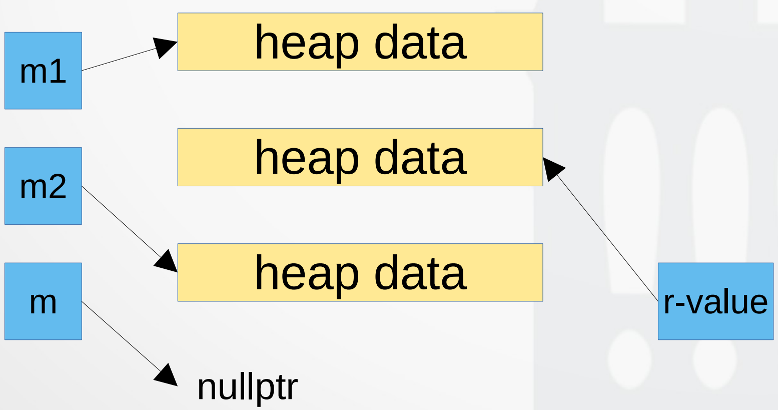
```
Matrix copyMatrix(Matrix m)
{
    return m;
}
```

r-value

Step 1: m1 is copied into m

Step 2: m is move copied into return r-value

Step 3: r-value is move assigned into m2



Move/Copy Constructors/Assignment

```
Matrix m1(2,2);
Matrix m2 = Matrix(4,4);

m2 = copyMatrix(m1);
```

```
Matrix copyMatrix(Matrix m)
{
    return m;
}
```

Step 1: m1 is copied into m

Step 2: m is move copied into return r-value

Step 3: r-value is move assigned into m2

Step 4: m and r-value go out of scope