CSCI 335
Software Design and Analysis III
Lecture 3a: Analysis of Algorithms

Professor Anita Raja

1

1

Agenda

- Big Five
- C++11: Templates/Matrices
- Asymptotic Analysis of Algorithms
- Assignment 1 Discussion

2

2

Announcements

- UTA Office Hours:
  - https://docs.google.com/document/d/1XPsurpIJR_20lxlHsLRMT40aBb1ldHh-f6c0Zv7bPzk/edit?usp=sharing
- Begin work on assignment.
- Verification of enrollment survey – deadline September 9, 2022

3

3

"The Big Five!" (not Three)

- Destructor
- Copy Constructor
- Copy Assignment operator =
- Move Constructor
- Move Assignment operator =
- When do defaults fail?
  - Shallow copy vs. deep copy

4

4

The Big Five: Simple Test Program

Match the following:

```
#include "IntCell.h"
void foo() {
  1. IntCell A{10};   //??
  2. IntCell B{A};    //??
  3. IntCell B=A      //??
  4. IntCell X = A;  // ??
  5. IntCell C;       // ??
  6. C=A;            //??
   IntCell *D;
  7. D = new IntCell; //??
  8. delete D; // ??
  9. IntCell A{move(B)}; // ??
  10. X = move(A); // ??
```

1. Zero-parameter constructor
2. One-parameter constructor
3. Copy constructor
4. Copy Assignment operator
5. Move constructor
6. Move assignment
7. Destructor

5

5

Modified IntCell to hold a pointer to an integer *sans "The Big Five"*

Class definition

```
class IntCell {
  public:
    explicit IntCell(int initial_value = 0)
      { stored_value_ = new int{initial_value};
    int Read() const
      { return *stored_value_; }
    void Write( int x )
      { *stored_value_ = x; }
  private:
    int *stored_value_;
};
```

Test Function

```
int TestFunction() {
    IntCell a{2}; //One-parameter constructor
    IntCell b = a; // Copy constructor
    IntCell c; // Zero-parameter constructor
    c = b; // Copy Assignment operator
    a.Write(4);
    cout << a.Read() << endl <<
b.Read( ) << endl << c.Read( )
<< endl;
    return 0;
}
```

7

7

Correct Implementation with "The Big Five"

```
// Destructor – stops shallow copying
IntCell::~IntCell( )
{
    delete stored_value_;
}
```

```
// Copy constructor
IntCell::IntCell(const IntCell & rhs)
{
    stored_value_ = new int{*rhs.stored_value_};
}
```

8

8

Correct Implementation with "The Big Five"

```
// Copy assignment operator – check for aliasing
IntCell & operator=(const IntCell & rhs)
{
    if (this != &rhs)
        *stored_value_ = *rhs.stored_value_; // assumes initial value
    return *this;
}
```

9

9

Move Constructor and Move Assignment

```
// Move constructor
IntCell(IntCell && rhs) :stored_value_{rhs.stored_value_}{
        rhs.stored_value_ = nullptr;
}


// Move assignment operator
IntCell & operator=(IntCell &&rhs) {
  // Use std::swap for all data members
    std::swap(stored_value_, rhs.stored_value_ );
    return *this;
}
```

10

10

Move Constructor and Move Assignment*

```
// Expand IntCell so that it contains a vector:
//   private: vector<int> items_; //i.e. non-primitive type

// Move constructor
IntCell(IntCell && rhs) :
stored_value_{rhs.stored_value_},
items_{std::move(rhs.items_) }{
         rhs.stored_value_ = nullptr;
}


// Move assignment operator
IntCell & operator=( IntCell && rhs ) {
  // Use std::swap for all data members
    std::swap(stored_value_, rhs.stored_value_);
    std::swap(items_, rhs.items_);
    return *this;
}
```

11

11

"The Big Five! – Final notes"

- Default behavior can be stated:
  - `IntCell(const IntCell &rhs) = default;`
- Or the function can be disabled:
  - `IntCell(const IntCell &rhs) = delete;`
- Normally, if copy-constructor is disabled, then assignment operator should also be disabled:

```
IntCell(const IntCell &rhs) = delete;
IntCell &operator=(const IntCell &rhs) = delete;
// If the above are deleted then, the expressions such as
// IntCell A = B; IntCell A{C}; … cause error.
```

If you implement one of the "big five", then you should implement all.

12

12

Exercise

- Fig01_16.cpp
- Fig01_18.cpp
- hw1_demo.png

13

13

Agenda

- Big Five
- C++11: Templates/Matrices
- Asymptotic Analysis of Algorithms
- Code demo

14

14

Templates

- Type-independent or generic algorithms
- Function templates
  - Pattern of what could become a function
  - (example FindMax, usage)
- Class templates
  - Pattern for any type of object.
  - (example IntCell, usage)

15

15

Function template for findMax

Function definition

```
// Return the maximum item in array a.
// Assumes a.size( ) > 0.
// Comparable objects must provide operator< and operator=
//
template <typename Comparable>
const Comparable & FindMax(const vector<Comparable> &a) {
   if (a.empty())
   abort();
   size_t max_index = 0;

   for (size_t i = 1; i < a.size( ); ++i)
      if (a[max_index] < a[ i ])
         max_index = i;
   return a[max_index ];
}
```

Main definition

```
#include <iostream>
#include <string>
#include "intCell.h"
using namespace std;

int main( ) { //code bloat
   vector<int>       v1( 37 );
   vector<double>  v2( 40 );
   vector<string>   v3( 80 );
   vector<IntCell>   v4( 75 );

 // Additional code to fill in the vectors not shown
   cout << FindMax( v1 ) << endl;  // OK?
   cout << FindMax( v2 ) << endl;  // OK?
   cout << FindMax( v3 ) << endl;  // OK?
   cout << FindMax( v4 ) << endl;  // OK?

   return 0;
}
```

16

16

Class template MemoryCell

```
/**
* A class for simulating a memory cell.
*/
template <typename Object>
class MemoryCell {
        public:
        explicit MemoryCell(const
                Object & initialValue=Object{})
                : storedValue{ initialValue }{}
        const Object & read( ) const {
                return storedValue; }
        void write( const Object & x ) {
                storedValue = x; }
        private:
        Object storedValue;
};
```

```
int main( ) {
MemoryCell<int> m1;
MemoryCell<string> m2{ "hello" };
m1.write( 37 );
m2.write( m2.read( ) + " world" );
cout << m1.read( ) << endl
<< m2.read( ) << endl;
return 0;
}
```

17

17

Object, Comparable

- Generic types used in this book/course.
- Object: at least
  - zero-parameter constructor
  - operator=
  - Copy constructor
- Comparable: at least
  - All of the above
  - operator<
- E.g. Fig01_23.cpp

18

18

Function Objects

- Limitation of templates
  - In the FindMax example, operator< needs to be defined for Comparable – Any problem?
- Idea:
  - Pass array of Objects AND a function that compares them
- How to pass a function as a parameter?
  - define a class with no data and one member function and pass an instance of the class
  - function is being passed by placing it inside an object (function object)
  - E.g. Fig 1.24, 1.25

19

19

FindMax Example revised

```
// Generic FindMax, with a function object, Version #1.
// Precondition: a.size( ) > 0.
template <typename Object, typename Comparator>
const Object & FindMax(const vector<Object> & arr, Comparator cmp) {
  if (arr.size() == 0) abort();
  size_t max_index = 0;

  for (size_t i = 1; i < arr.size( ); ++i)
    if (cmp.IsLessThan(arr[ maxIndex ], arr[ i ]))
        max_index = i;

  return arr[max_index];
}
```

20

20

FindMax Example revised

```
// Generic FindMax, with a function object, Version #1…continued…
class CaseInsensitiveCompare {  // Comparator 1.
  public:
    bool IsLessThan(const string & lhs, const string & rhs) const
      { return stricmp( lhs.c_str( ), rhs.c_str( ) ) < 0; }
};

class YetAnotherCompare {  // Comparator 2.
  public:
    bool IsLessThan(const string & lhs, const string & rhs) const
      { if (lhs.length() == rhs.length()) return lhs < rhs;
        else return  lhs.length() < rhs.length();
      }
};
int main( )
{
    vector<string> arr( 3 );
    arr[0] = "ZEBRA"; arr[1] = "alligator"; arr[2] = "crocodile";
    cout << FindMax(arr, CaseInsensitiveCompare{}) << endl; //answer is ZEBRA
    cout << FindMax(arr, YetAnotherCompare{}) << endl; ; //answer is ZEBRA
    return 0;
}
```

21

21

FindMax Example revised: operator overloading

```
// Generic FindMax, with a function object, C++ style.
// Precondition: a.size( ) > 0.
template <typename Object, typename Comparator>
const Object & findMax( const vector<Object> & arr, Comparator IsLessThan) {
   if (a.size() == 0) abort();
   size_t max_index = 0;

   for (size_t i = 1; i < arr.size( ); ++i)
      if (IsLessThan( arr[ maxIndex ], arr[ i ]))
         max_index = i;
   return arr[max_index];
}

// Generic FindMax, using default ordering.
#include <functional>
template <typename Object>
const Object &FindMax(const vector<Object> & arr) {
   return FindMax(arr, less<Object>{ });
}
```

22

22

FindMax Example revised

```
// Generic findMax, with a function object, C++ style… cont …
class CaseInsensitiveCompare {  // Comparator 1, C++ style.
  public:
    bool operator( ) (const string & lhs, const string & rhs) const
      { //… Same as in slide 21.}
};
class YetAnotherCompare {  // Comparator 2, C++ style.
  public:
    bool operator( ) (const string & lhs, const string & rhs) const
      { //… Same as in slide 21. }
};
int main( ) {
    vector<string> arr(3);
    arr[0] = "ZEBRA"; arr[ 1 ] = "alligator"; arr[ 2 ] = "crocodile";
    cout << FindMax(arr, CaseInsensitiveCompare{}) << endl; ; //answer is ZEBRA
                                                              //and then crocodile
    cout << FindMax(arr, YetAnotherCompare{}) << endl;
    cout << FindMax(arr) << endl;
    return 0;
}
```

23

23

Matrix

- A 3 by 4 matrix of strings is something like…

"a"	"b"	"dd"	"ee"
"1"	"2"	"cc"	"dd"
"aa"	"e"	"f"	"gg"

- If M is a matrix we may want to access the rows.
  For instance:
  M[0] = { "a", "b", "dd", "ee"}
  M[2] = {"aa", "e", "f", "gg"}
  M[3] .... We need to raise an exception here….
- We want to construct and destruct as well.

24

24

Matrix Example

```
#ifndef MATRIX_H_
#define MATRIX_H_
#include <vector>
namespace my_linear_algebra {
 template <typename Object>
 class Matrix {
   public:
     Matrix(int rows, int cols) : matrix_2d_(rows) {
        for (auto &this_row: matrix_2d_)
           matrix_2d_[ i ].resize( cols );  }
     Matrix(std::vector<std::vector<Object>> v): matrix_2d_{v} { }
     Matrix(std::vector<std::vector<Object>> &&v): matrix_2d_{std::move(v)} { }
     const std::vector<Object> & operator[](int row) const
       { return matrix_2d_[ row ]; }
     std::vector<Object> & operator[](int row)
       { return matrix_2d_[ row ]; }
     int NumRows( ) const
       { return matrix_2d_.size( ); }
     int NumCols( ) const
       { return (NumRows( ) != 0) ? matrix_2d_[0].size( ) : 0; }
   private:
     std::vector<std::vector<Object>> matrix_2d_;
 };
 } // namespace my_linear_algebra
 #endif // MATRIX_H_
```

constructor

25

25

Matrix Example

```
#ifndef MATRIX_H_
#define MATRIX_H_
#include <vector>
namespace my_linear_algebra {
 template <typename Object>
 class Matrix {
   public:
     …

    const std::vector<Object> & operator[](int row) const
      { return matrix_2d_[ row ]; }

    std::vector<Object> & operator[](int row)
      { return matrix_2d_[ row ]; }

      …

private:
    std::vector<std::vector<Object>> matrix_2d_;
};
} // namespace my_linear_algebra
#endif  // MATRIX_H_
```

26

26

Matrix

- Copying matrices

```
// @from: an input matrix.
// @to: output matrix.
// Matrix @from will be copied to @to.
// We assume that @from and @to have the same size.
 void CopyToMatrix(const Matrix<int> &from, Matrix<int> &to) {
        // Add code to check whether @from/@to are of the same size.
        // If not through exception.
             for (int = 0; i < to.NumRows(); i++)
                  to[i] = from[i];
 }
```

27

27

Chapter 1 Summary

- Lvalues, Rvalues, References
- Swap-and-copy
- Big Five
- Parameter and Return Passing
- Templates
- Matrices

28