CSCI 335 Software Design and Analysis III

Class Mechanics & Introduction (Chapter 1)

Ioannis Stamos

Introduction

- Instructor
  - Ioannis Stamos, Professor
    - PhD from Columbia University.
    - Teaches at Hunter & the Graduate Center
  - http://www.cs.hunter.cuny.edu/~ioannis
  - Director of Computer Vision laboratory
- Syllabus & Programming Rules
  - How to Submit your Assignments

C++ review with emphasis on C++11concepts

- Programming Style is very important
  - Code Readability
  - Code Reuse
  - Code Maintenance
- Good source: Google C++ style guide.

C++ review with emphasis on C++11concepts

- Examples
  - Class names *CamelCase* : class PointFinder
  - Function names *CamelCase*: FindClosestPoint(…)
  - Local variables and parameters (lower case, separate words with _): int number_of_neighbors;
  - Example function:

```
double AbsDistance(const array<double, 3> &point1,
                   const array<double, 3> &point2,
                   bool use_two_params_only = false) {
  const double result2 = fabs(point1[0] - point2[0]) + fabs(point1[1] - point2[1]);
  return use_two_params_only ? result2:
                               result2 + fabs(point1[2] - point2[2]);
}
```

C++ review with emphasis on C++11concepts

```
// @point1: a 3D point.
// @point2: a 3D point.
// @use_two_params_only: if true, use only first two coordinates.
// Compute and return the absolute distance between the points.
double AbsDistance(const array<double, 3> &point1,
                   const array<double, 3> &point2,
                   bool use_two_params_only = false) {
  const double result2 = fabs(point1[0] - point2[0]) + fabs(point1[1] - point2[1]);
  return use_two_params_only ? result2:
                               result2 + fabs(point1[2] - point2[2]);
}
```

C++ review with emphasis on C++11concepts

- Class names / types: CamelCase.
- Public/private class data members (lower case and underscore at end): a_variable_
- Constants as class members: const double kInitialValue = 10.0;
- Functions: CamelCase.
- Place public before private.

```
class PointFinder {
  public:
    const double kInitialValue = 10.0;
    PointFinder() {};
    double AbsDistance(const std::array<double, 3> &a_point);
    std::array<double, 3> FindClosestPoint(std::array<double, 3> input_point,
                                           double max_distance_from_point);
  private:
    std::array<double, 3> initial_point_;
}
```

- Use full names that describe what the variable is doing. Example:
  - double minimum_distance_from_start;
  - // Do not use cryptic and small names: double fg;
  - Indices in loops can be i, j, k, etc.

C++ review with emphasis on C++11concepts

- Comments
  - In start of file providing a brief description of the contents of file and author name.

C++ review with emphasis on C++11concepts

- Comments
  - **On top of class names providing a brief description:**

```
// Search for closest point from a given set of 3D points.
// Sample usage:
// PointFinder a_finder;
// a_finder.AddPoint(std::array<double, 3>{1.0, 3.0, 10.0});
// auto closest_point = a_finder.FindClosestPoint(std::array<double, 3>{0,1,0}, 30.0);
class PointFinder { … }
```

C++ review with emphasis on C++11concepts

- Comments
  - On top of each function (unless functionality is obvious). In case of class functions, put comment only in header (do not replicate).

```
// @input_point: A given 3D point.
// @max_distance_from_point: Do not look beyond that distance for a closest  point.
// @return the closest point to input_point.
std::array<double, 3> FindClosestPoint(std::array<double, 3> input_point, double
                                       max_distance_from_point);
```

  - Inside the implementations, only if it is hard to understand from the code.

C++ classes

- IntCell class (just to hold one integer)
  - Default parameter
- Initializer List example
  - Explicit constructor
- Accessor member functions
- Mutator member functions

```cpp
//
// A class for simulating an integer memory cell.
//
class IntCell {
  public:
    explicit IntCell(int initial_value = 0): stored_value_{initial_value} { }

    int Read() const
      { return stored_value_; }

    void Write(int x)
      { stored_value_ = x; }

  private:
    int stored_value_;
};
```

C++11 Initialization

- On the previous slide we wrote

```
stored_value_{initial_value}
```

- Instead of

```
stored_value_(initial_value)
```

- This is part of a larger effort to provide a uniform syntax for initialization.
- Generally speaking, anywhere you can initialize, you can do so by enclosing initialization in braces.

C++11 Initialization

- Correct:

```
IntCell obj1;      // Zero-parameter constructor
IntCell obj2(12); // One-parameter constructor (<= C++11)
```

- Incorrect in C++, inconsistent with initializer list syntax:

```
IntCell obj4();     // Function declaration
```

- Incorrect in C++, inconsistent with explicit constructor:

```
IntCell obj3 = 35
```

- Correct in C++11, now consistent with use in initializer lists:

```
IntCell obj2{12}; // One-parameter, as before
IntCell obj4{};   // Zero-parameter, as before
```

C++11 Vector Initialization

- It is now possible to write:
  - `vector<int> numbers = {1, 2, 3, 4};`
  - `vector<int> numbers {1, 2, 3, 4};`
- Consider this:
  - `vector<int> a(12);`
  - `vector<int> a{12}; // ?`
- Should this be a vector of size 12 or a vector of size 1 with the value 12 in position 0?

C++11 Range For Loops

```
vector<float> some_numbers{1.1, 10.2, 3, 20.31};
// Compute their sum.
float sum = 0;
// “Old” C++ way:
for (size_t i = 0; i < some_numbers.size(); ++i)
    sum += some_numbers[i];
```

C++11 Range For Loops

```
// New way - range loop.
vector<float> some_numbers{1.1, 10.2, 3, 20.31};

float sum = 0;
for (float x : some_numbers) {
    sum += x;
}
```

- This loop is only appropriate when
  - accessing elements sequentially and
  - when the index is not needed.
- Note: x cannot be modified here.

C++11 Range For Loops and auto

```
vector<float> some_numbers{1.1, 10.2, 3, 20.31};

float sum = 0;
for (auto x: some_numbers) {
    sum += x;
}
```

- `auto` keywords signifies that compiler determines type

C++ details

- Pointers (anybody?)
- Dynamic object declaration
- Garbage collection and delete
  - Memory leak if you are not careful
    - Do not dynamically allocate memory unless you have to!

```cpp
//Dynamic object declaration example

 int main() {
     IntCell *m = nullptr; // C++11 null pointer literal.

     m = new InteCell{}; // C++11.
     // m = new InteCell; // OK. This textbook.
     // m = new IntCell();  // Still OK
     m->Write(5);
     cout << "Cell contents: " << m->Read() << endl;

     delete m;
     return 0;
 }
```

C++11 Lvalues and Rvalues

- **Lvalue**:

  expression that identifies a non-temporary object.
- **Rvalue**: expression that
  - identifies a temporary object OR
  - is a value not associated with an object (literal).
- A function can return an Lvalue or Rvalue.
- A function's parameter can be an Lvalue or Rvalue.

C++11 Lvalues and Rvalues

```
const int x = 2;
int y;
int z = x + y;
vector<string> arr(3);
string str = "foo";
vector<string> *ptr = &arr;
```

C++11 Lvalues and Rvalues

```
const int x = 2;
int y;
int z = x + y;
vector<string> arr(3);
string str = “foo”;
vector<string> *ptr = &arr;
```

- **Lvalues**: x, y, and z, since they are named expressions. Same for arr, str, ptr.
- **Rvalues**: 2 and x + y, since 2 is a literal and x + y is a temporary value. Same for 3 and “foo”.

Lvalue reference (= synonym)

string str = “fine”;
// Lvalue reference:
string &rstr = str; // rstr another name of str.

rstr += ‘o’; // changes str?
cout << (&str == &rstr) << endl; // True or False?
string **&b1** = “hello”; // legal ?
string **&b2** = str + “a”; // legal ?
string **&sub** = str.substr(0, 4); // legal ?

Rvalue reference

```
string str = “fine”;
// Rvalue references:
string &&b1 = “hello”; // OK.
string &&b2 = str + “a”;  // OK.
string &&sub = str.substr(0,4); // OK.
```

- Why? Move semantics. Stay tuned…

Lvalue Reference Use #1

Simplifying complicated expressions

Example:

```
size_t ConvertFirstLetter(const string &string_1) {
   return string_1.empty() ? 0 : static_cast<size_t>(string_1[0]);
}
vector<list<string>> a_vector_of_lists_of_strings;
```

Lvalue Reference Use #1

Simplifying complicated expressions

```
----------------------------------------

const string name = “bottle”;

auto &which_list =
a_vector_of_lists_of_strings[ConvertFirstLetter(name)];

which_list.push_back(x);
```

Lvalue Reference Use #1

Simplifying complicated expressions

```
const string name = "bottle";

auto &which_list =
a_vector_of_lists_of_strings[ConvertFirstLetter(name)];

which_list.push_back(x);

[auto can be replaced with list<string>]
```

Lvalue Reference Use #2

Making changes in range for loops

```
vector<int> a_vector{10, 3, 4};
for (auto &x : a_vector) ++x;

// Old way:
// for (int size_t = 0; i < a_vector.size(); ++i)
//      ++a_vector[i];
```

Lvalue Reference Use #2

Making changes in range for loops

```
vector<int> a_vector{10, 3, 4};
for (auto  x: a_vector) ++x;
// No reference used for x.
// What is the result?
```

Lvalue Reference Use #3

- *Avoiding a copy*

```
vector <string> a_vector{“a”, “zebra”, “name”};
------------OPTION 1-----------------
string FindMax1(const vector<string> &arr) {…}
string result = FindMax1(a_vector);

------------OPTION 2-----------------
const string &FindMax2(const vector<string> &arr) {…}
// i.e. FindMax2() returns a non-modifiable reference.
const string &result = FindMax2(arr);
```

```cpp
//----OPTION 1-----
// @arr: A non-empty vector of strings.
// @return the maximum string in the @arr.
//  Will abort() if @arr is empty.
string FindMax1(const vector<string> &arr) {
   if (arr.empty()) abort();
   int max_index = 0;
   for (int i = 1; i < arr.size(); ++i)
        if( arr[max_index] < arr[i] )
            max_index = i;
    return arr[max_index];
}
```

```
//----OPTION 2-----
// @arr: A non-empty vector of strings.
// @return the maximum string in the @arr.
//  Will abort() if @arr is empty.
const string &FindMax2(const vector<string> &arr) {
    if (arr.empty()) abort();
    int max_index = 0;
    for (int i = 1; i < arr.size(); ++i)
        if( arr[max_index] < arr[i] )
            max_index = i;
    return arr[max_index];
}
```

C++ details

- Parameter passing

```
double ComputeAverageAndOffset(const vector<int> &arr,
                               double offset,
                               bool &error_flag) {
   double the_average = 0.0;
   for (const auto &x: arr) the_average += x;
   if (!arr.empty()) the_average /= static_cast<double>(arr.size());
   error_flag = arr.empty();
   return the_average – offset;
}
```

  - Call by value
    - Small objects that will not be changed by function
  - Call by reference
    - Objects that may be changed by function
  - Call by constant reference
    - Large objects that will not be changed by function

C++11 Call by Rvalue Reference

- Rvalue stores a temporary value.
- Functions can treat it as such or not.
- But function "knows" if it is a temporary or not based on signature.

Example:

```
string RandomItem(const vector<string> &arr) {
    cout << "Version 1" << endl;
    const size_t n = std::rand() % arr.size();
    return arr[n];
}

string RandomItem(vector<string> &&arr) {
    cout << "Version 2" << endl;
    const size_t n = std::rand() % arr.size();
    return arr[n];
}
```

C++11 Call by Rvalue Reference

```
const vector<string> v{"hello", "world”};

// Which Version?
cout << RandomItem(v) << endl;

// Which Version?
cout << RandomItem({"hello", "world”}) << endl;
```

Return Passing

- Return by
  - Value
  - Constant reference
  - Reference

- In C++11, return by value may be efficient even for large objects if the returned object is an Rvalue.

Return by
Value vs Constant Reference

```
//--------OPTION 1: Return by value---------------
template<typename Object>
Object RandomItem1(const vector<Object> &arr)
{ return arr[std::rand() % arr.size()]; }


//--------OPTION 2: Return by constant reference---------------
template<typename Object>
const Object &RandomItem2(const vector<Object> &arr)
{ return arr[std::rand() % arr.size()]; }
```

Return by
Value vs Const Reference

```
class LargeType {... private: vector<list<vector<string>>> p_; }
vector<LargeType> vec;

LargeType item1 = RandomItem1(vec);           // copy.
LargeType item2 = RandomItem2(vec);           // copy.
const LargeType &item3 = RandomItem2(vec);   // no copy.
auto &item4 = RandomItem2(vec);              // no copy.
```

Returning stack-allocated Rvalue (C++ 11) => efficient now

```
vector<int> PartialSum(const vector<int> &arr) {
    if (arr.empty()) abort();
    vector<int> result(arr.size());
    result[0] = arr[0];
    for(size_t i = 1; i < arr.size(); ++i)
        result[i] = result[i - 1] + arr[i];
    return result;
}

vector<int> sums = PartialSum(vec);
// Copy in old C++; move in C++11
// Efficient in C++11.
// You can also write:
auto sums = PartialSum(vec);
```

C++11 std::swap and std::move

Copying large objects is expensive, if the object's class supports move then we can be more efficient.

- STL containers (like vector) now support move.
- Move can be used by casting the right-hand side of an assignment to an Rvalue reference.

```
// x is an object of type vector<string>

vector<string> tmp =
static_cast<vector<string> &&>(x);
```

- The above code is equivalent to

```
vector<string> tmp = std::move(x);
```

C++11 std::swap and std::move

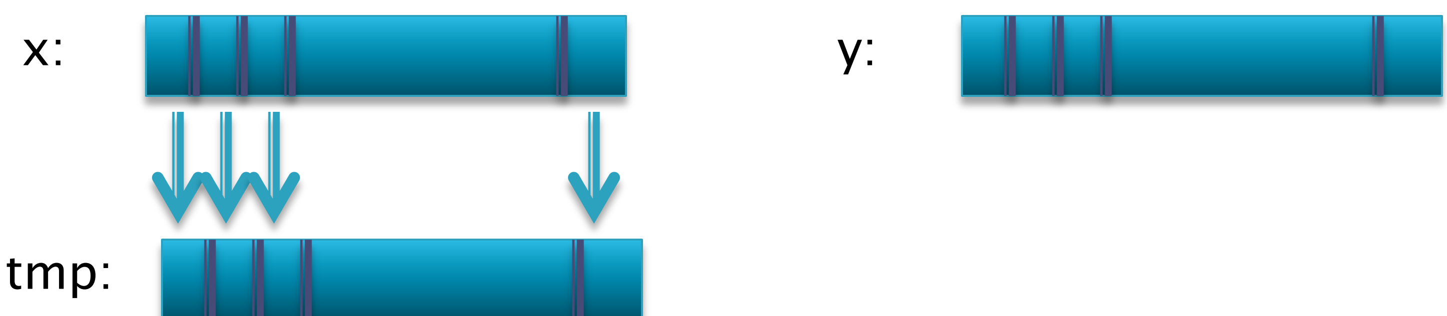
```
// Swap by three copies
void swap(vector<string> &x, vector<string> &y) {
    vector<string> tmp = x;
    x = y;
    y = tmp;
 }
```

C++11 std::swap and std::move

```
// Swap by three copies
void swap(vector<string> &x, vector<string> &y) {
    vector<string> tmp = x;
    x = y;
    y = tmp;
 }
```

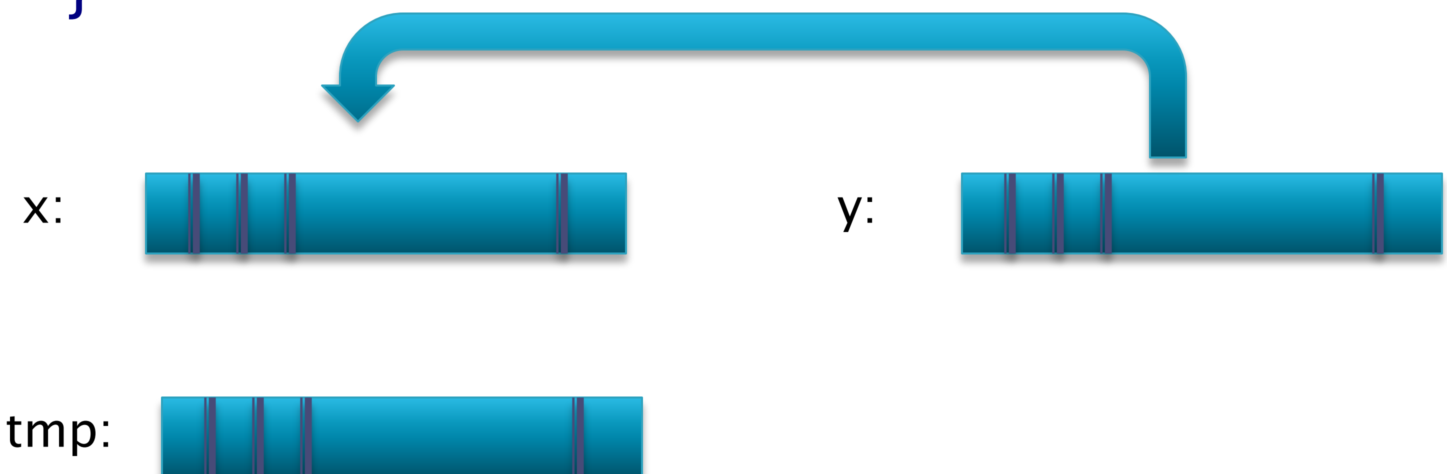
C++11 std::swap and std::move

```
// Swap by three copies
void swap(vector<string> &x, vector<string> &y) {
    vector<string> tmp = x;
    x = y;
    y = tmp;
 }
```



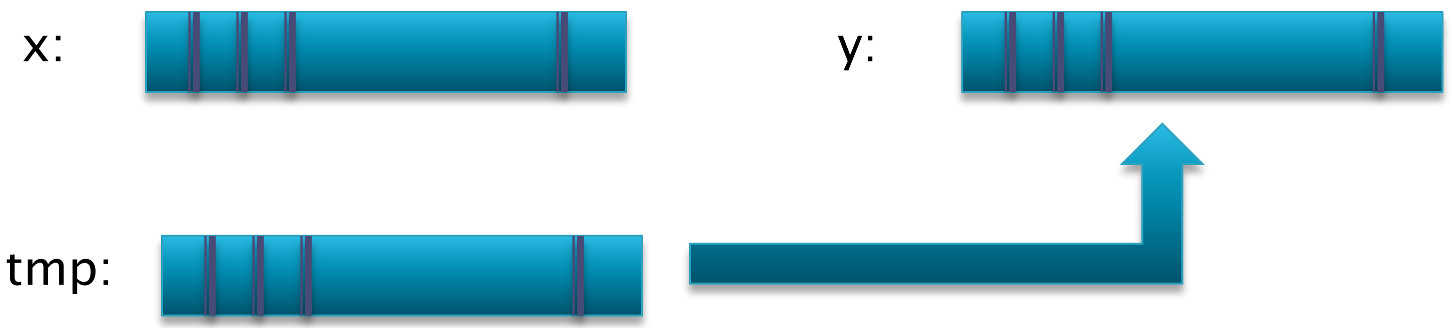
C++11 std::swap and std::move

```
// Swap by three copies
void swap(vector<string> &x, vector<string> &y) {
    vector<string> tmp = x;
    x = y;
    y = tmp;
 }
```



C++11 std::swap and std::move

```
// Swap by three copies
void swap(vector<string> &x, vector<string> &y) {
    vector<string> tmp = x;
    x = y;
    y = tmp;
 }
```



C++11 std::swap and std::move

```
// std::move() is just a type-cast
// std::move() just converts an Lvalue to an Rvalue
// More efficient: Swap by three moves
void swap(vector<string> &x, vector<string> &y) {
    vector<string> tmp = std::move(x);
    x = std::move(y);
    y = std::move(tmp);
 }

// std::swap is now part of the STL and works for any type
// So you don't have to implement the above for STL types
// You can write:
//   vector<string> x;
//   vector<string> y;
//   std::swap(x,y);
```

C++11 std::swap and std::move

```
// std::move() is just a type-cast
// std::move() just converts an Lvalue to an Rvalue
// More efficient: Swap by three moves
void swap(vector<string> &x, vector<string> &y) {
    vector<string> tmp = std::move(x);
    x = std::move(y);
    y = std::move(tmp);
}
```

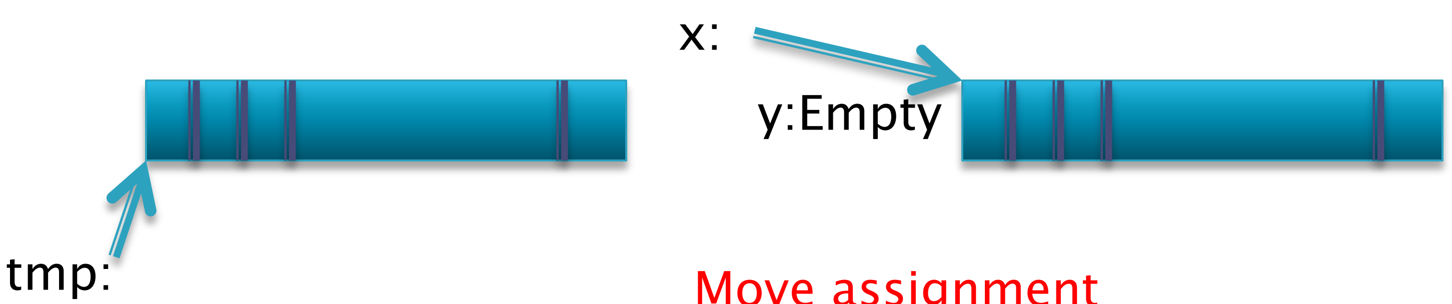
C++11 std::swap and std::move

```
// std::move() is just a type-cast
// std::move() just converts an Lvalue to an Rvalue
// More efficient: Swap by three moves
void swap(vector<string> &x, vector<string> &y) {
    vector<string> tmp = std::move(x);
    x = std::move(y);
    y = std::move(tmp);
 }
```

Move assignment

C++11 std::swap and std::move

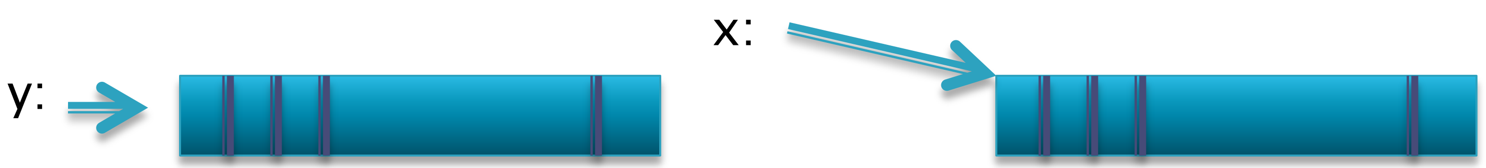
```
// std::move() is just a type-cast
// std::move() just converts an Lvalue to an Rvalue
// More efficient: Swap by three moves
void swap(vector<string> &x, vector<string> &y) {
    vector<string> tmp = std::move(x);
    x = std::move(y);
    y = std::move(tmp);
 }
```



Move assignment

C++11 std::swap and std::move

```
// std::move() is just a type-cast
// std::move() just converts an Lvalue to an Rvalue
// More efficient: Swap by three moves
void swap(vector<string> &x, vector<string> &y) {
    vector<string> tmp = std::move(x);
    x = std::move(y);
    y = std::move(tmp);
 }
```



tmp:Empty

Move assignment

“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

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

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

Modified IntCell to hold **a pointer** to an integer ***without*** "***The Big Five***"

```
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_;
};
```

```
int TestFunction() {
    IntCell a{2};
    IntCell b = a;
    IntCell c;

    c = b;
    a.Write(4);
    cout << a.Read() << endl << b.Read( ) << endl << c.Read( ) << endl;
    return 0;
}
```

Correct Implementation with "The Big Five"

--

```
// Destructor
IntCell::~IntCell( )
{
   …
}

// Copy constructor
IntCell::IntCell(const IntCell & rhs)
{
   …
}

// Copy assignment operator
IntCell & IntCell::operator=(const IntCell & rhs)
{
   …
}
```

Correct Implementation with “The Big Five”

```cpp
// Destructor
IntCell::~IntCell( )
{
    delete stored_value_;
}

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

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

Move Constructor and Move Assignment

```
// Move constructor
IntCell(IntCell && rhs) {
    …
}

// Move assignment operator
IntCell & operator=( IntCell && rhs ) {
    …
}
```

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) {
    …
}
```

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;
}
```

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;
}
```

C++11 Style Copy Assignment

// Copy-and-swap idiom.
// In C++11 this is the usual implementation

```
IntCell & operator=( const IntCell &rhs ) {
      IntCell copy = rhs;  // Calls the copy-constructor
      std::swap(*this, copy);
      return *this;
  }
```

IMPORTANT NOTE:

- If swap is implemented using copy assignments, there will be a mutual **non-terminating recursion**.

  => swap should be implemented either

with three moves or swapping data member by data member.

“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.**

Templates

- Type-independent or generic algorithms
- Function templates
  - (example FindMax, usage)
- Class templates
  - (example IntCell, usage)

```
//
// 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 ];
}
```

```
#include <iostream>
#include <string>
#include “intCell.h”
using namespace std;
int main( ) {
   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;
}
```

Object, Comparable

- Generic types used in this book
- Object: at least
  - zero-parameter constructor
  - operator=
  - Copy constructor
- Comparable: at least
  - All of the above
  - operator<
- Example

```
class Employee {
  public:
   // ... Big-5 not shown...
    void SetValue(const string & n, double s)
      { name = n; salary = s; }

    const string & name( ) const
      { return name_; }
    void Print(ostream & out) const
      { out << name_ << " (" << salary_ << ")"; }
    bool operator<(const Employee &rhs) const
      { return salary_ < rhs.salary_; }


  private:
    string name_;
    double salary_;
};
```

```
// Define an output operator for Employee. It is a standalone
//  function, outsize of class Employee.
ostream &operator<<( ostream & out, const Employee & rhs) {
    rhs.Print( out );
    return out;
}

int main( ) {
    vector<Employee> v( 3 );

    v[0].setValue( “John Adams", 400000.00 );
    v[1].setValue( “Bill G", 2000000000.00 );
    v[2].setValue( “X Y", 13000000.00 );

     cout << findMax( v ) << endl;


    return 0;
}
```

Adding a template object

```
template <typename Object>
class Employee {
  public:
    // Big-5 not shown.
    void SetValue(const string & n, double s, const Object &other=Object{})
     { name_ = n; salary_ = s; other_= other; }

  const string & name( ) const
     { return name_; }
    void Print( ostream & out ) const
     { out << name_ << " (" << salary << ")"; out << other_; }
    bool operator<( const Employee & rhs ) const
     { return salary_ < rhs.salary_; }

  private:
    string name_;
    double salary_;
    Object other_;
};
```

Adding a template object

```
 // Define an output operator for Employee
template <typename Object>
ostream & operator<<(ostream & out, const Employee<Object> & rhs) {
    rhs.print(out);
    return out;
}

int main( ) {
    vector<Employee<string>> v(3);

    v[0].setValue( “John Adams", 400000.00, “comment 1” );
    v[1].setValue( “Bill G", 2000000000.00, “comment 2” );
    v[2].setValue( “X Y", 13000000.00, “comment 3” );

    cout << v[1] << endl; //Will this work?

    return 0;
}
```

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?
    - Pass a class that contains only one member.
    - Function object
  - Implementation 1
  - Implementation 2

```
// 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];
}
```

```
// 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;
    cout << FindMax(arr, YetAnotherCompare{}) << endl;
    return 0;
}
```

```
// 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>{ });
}
```

```
// 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 55.}
};
class YetAnotherCompare {  // Comparator 2, C++ style.
  public:
    bool operator( ) (const string & lhs, const string & rhs) const
      { //... Same as in slide 55. }
};

int main( ) {
    vector<string> arr(3);
    arr[0] = "ZEBRA"; arr[ 1 ] = "alligator"; arr[ 2 ] = "crocodile";
    cout << FindMax(arr, CaseInsensitiveCompare{}) << endl;
    cout << FindMax(arr, YetAnotherCompare{}) << endl;
    cout << FindMax(arr) << endl;

    return 0;
}
```

Matrix

- Code that implements a matrix…
- A 3 by 4 matrix of strings is something like…

$$\begin{bmatrix} \text{“a”} & \text{“b”} & \text{“dd”} & \text{“ee”} \\ \text{“1”} & \text{“2”} & \text{“cc”} & \text{“dd”} \\ \text{“aa”} & \text{“e”} & \text{“f”} & \text{“gg”} \end{bmatrix}$$

- 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

```
#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_
```

```
#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)} { }
      ...

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

```
#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_
```

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) {
      …
}
```

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];
}
```