CSCI 335
Software Design and Analysis III
Lecture 2: C++11

Professor Anita Raja

1

Agenda

- C++
  - Lvalues and Rvalues
  - Parameter Passing
  - Return Passing
  - std::swap and std::move
  - Big Five
- Next lecture conclude C++:
  - Templates
  - Matrices

2

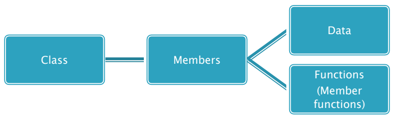
Announcements

- By tonight
  - TA office hours posted.
  - HW1 announcement posted on blackboard. Code available via github. Submissions only via gradescope.

3

C++ Classes: Basic class syntax

- Each instance of a class is an **object**.
- Each **object** contains the data components specified in the class.
- A **member function** is used to act on an object.
- Member functions are often called **methods**.



4

C++ classes

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

5

5

IntCell class

```
// 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_;
};
```

6

6

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.

7

7

C++11 explicit constructor

```
//Incorrect
  Intcell obj;   //obj is an IntCell
  obj = 37;      //should not compile; type mismatch

  //Usually compiler would attempt to convert
  obj = 37;
  //into
  IntCell temp = 37;
  obj = temp;
  //But explicit means that a one-parameter constructor cannot be used to
  generate an implicit temporary. Hence the compiler complaint.
```

8

8

C++11 Initialization: Objects are declared like primitivetypes

//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
```

9

9

C++11 Vector Initialization

- Vector and string classes in the STL treat arrays and strings as first-class objects.
- 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?

10

10

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];

// New way – range loop.;
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.

11

11

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` keyword signifies that compiler determines type

12

12

C++ Details

- POINTERS
- DYNAMIC OBJECT DECLARATION
- GARBAGE COLLECTION AND DELETE

13

Pointers Example

```
//Dynamic object declaration example
int main() {
IntCell *m = nullptr;  // C++11 null pointer literal.

m = new IntCell{}; // C++11.
// m = new IntCell; // Ok preferred in textbook.
// m = new IntCell();  // Still OK
m->write(5);
cout << "Cell contents: " << m->read() << endl;

delete m;
return 0;
}
```

14

14

C++11 Lvalues, Rvalues and References

- **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.

15

15

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;
```

16

16

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".

17

17

Lvalue reference (= synonym)

In C++11 lvalue reference declared by placing an & after some type.

string str = "human";
// Lvalue reference:
string &rstr = str; // rstr another name of str.

rstr += 'e'; // changes str?
cout << (&str == &rstr) << endl; // True or False?
string **&b1** = "humane"; // legal ?
string **&b2** = str + ""; // legal ?
string **&sub** = str.substr(0, 4); // legal ?

18

18

Rvalue reference

Rvalue ref declared by placing && after some type.

string str = "human";
// Rvalue references:

```
string &&b1 = "humane"; // OK.
string &&b2 = str + "";  // OK.
string &&sub = str.substr(0,4); // OK.
```

- Why? Move semantics. Stay tuned…

19

19

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;
```

20

20

Lvalue Reference Use #1

Simplifying complicated expressions

```
const string name = "bottle";

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


21

Lvalue Reference Use #1

Simplifying complicated expressions: Example 2

```
auto * whichList – theLists [myhash(x, the Lists.size())];
If (find( begin( which List ), end( whichList 0, x ), x ), x !=
end(whichList ) )
Return false;
which_list.push_back(x);
```

22

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);
```


23

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>]
```


24

Lvalue Reference Use #2

Making changes in "range for loops"

//**Goal :** Increment all values in vector by 1

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

//alt attempt with range for loop but broken

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

25

25

Lvalue Reference Use #2

Making changes in range for loops

//**correct**

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

26

26

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);
```

27

27

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

28

28

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

29

29

Lvalue Reference Use #3

- **Avoiding a copy**

1. Reference variables are often used to avoid copying objects across function-call boundaries (either in the function call or the function return)
2. Syntax is needed in function declarations and returns to enable the passing and returning using references instead of copies.

30

30

C++ details: Parameter Passing

- C and Java use call-by-value:
  - actual argument is copied into the formal parameter.
- C++ (large complex objects):
  - copying is inefficient and value may need to be changed.
- C++ has 3 ways to pass parameters
  - Call-by-value: Small objects that will not be changed by function.
  - Call-by-reference: All objects that may be changed by function.
  - Call-by-constant-reference : Large objects that will not be changed by function and are expensive to copy.

31

Why call-by-value is insufficient?

double average(double a, double b); //returns average of a and b
void swap(double a, double b); //swaps a and b; wrong parameter types
String randomItem(vector<string> arr); //returns a random item in arr;
//inefficient

Ideal use of call-by-value:
double z = average (x,y); //why?

Correct usage:
void swap(double &a, double &b); //call-by-reference
String randomItem(const vector<string> & arr); //call-by-constant-reference

32

C++11: Call by Rvalue Reference

- 4th way: Rvalue stores a temporary value.
  - x=rval (rval is rvalue) implemented by a move instead of a copy
  - Moving an object's state easier (simple pointer change vs copying it)
  - x=y //copy if y is lvalue
  - x=y //move if y is rvalue
- But function "knows" if it is a temporary or not based on signature.

33

2 versions: overloading a function

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

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

34

Return Passing

double average (double a, double b); //return average of a and b

largeType randomItem(const vector<LargeType> & arr);
//potentially inefficient

vector<int> partialSum(const vector<int> & arr);
//inefficient in C++11

- Uses return-by-value:
  - function returns an object of an appropriate type that can be used by caller.
  - In all cases, result is an Rvalue

36

Return Passing

- Return by
  - Value
  - Constant reference
  - Reference
- Read Section 1.5.4 for examples
- In C++11, return by value may be efficient even for large objects if the returned object is an Rvalue.

37

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) 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);
```

41

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

```
void  swap( double & x, double & y)
{
   double tmp = x;
   x = y;
   y= tmp;
}

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

42

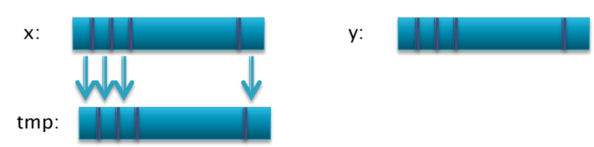
C++11 std::swap and std::move

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

43

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

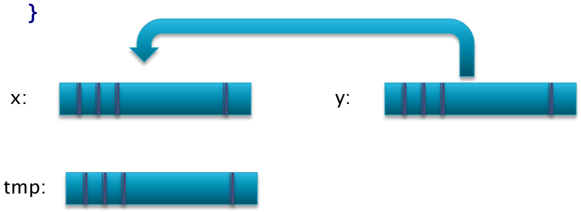
```
// Swap by three copies
void swap(vector<string> &x, vector<string> &y) {
    vector<string> tmp = x;
    x = y;
    y = tmp;
 }
```



44

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

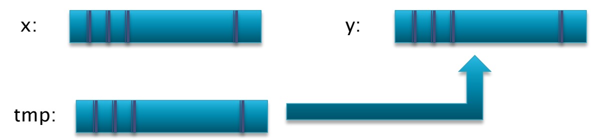
```
// Swap by three copies
void swap(vector<string> &x, vector<string> &y) {
    vector<string> tmp = x;
    x = y;
    y = tmp;
 }
```



45

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;
 }
```



46

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);
```

std::move converts any lvalue to rvalue.

Note: It doesn't literally move anything, just makes a value subject to be moved.

47

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);
 }
```

48

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

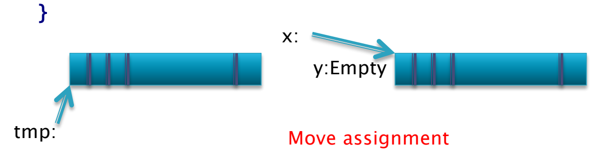
```
// std::move() is just a type-cast
// std::move() just converts an Lvalue to an Rvalue
// More efficient: Swap by three moves
void swap(vector<string> &x, vector<string> &y) {
    vector<string> tmp = std::move(x);
    x = std::move(y);
    y = std::move(tmp);
 }
```

Move assignment

49

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);
 }
```

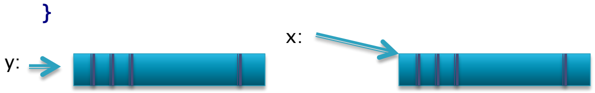


Move assignment

50

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

51