Object-Oriented Programming

rValue References

Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

lvalue

An expression that refers to a memory location, and therefore, we can get the address via the & operator

- Left side of an assignment

```
area = height * width
```

The *lvalue* is `area`

- An lvalue is required as the target of an assignment

An expression that is not a lvalue

rvalue

- Right side of an assignment

```
area = height * width
```

The *rvalue* is `height * width`

- An rvalue cannot be the target of an assignment

Example

```
int& foo();
int bar();

int main() {

    // lvalue examples
    int i = 42;
    i = 43;
    // i is an lvalue
    int* p = &i;
    // foo() is an lvalue
    foo() = 42;
    int* p1 = &foo();

    // rvalue examples
    // bar() is an rvalue
    i = bar();
    // int* p2 = &bar();
    // 42 is an rvalue
    i = 42;

    return 0;
}
```

- `foo()` returns an *lvalue*
- `bar()` returns an *rvalue*
- `i` is an *lvalue*
- A call to `foo()` is an *lvalue*
- Can point to the result of a call to `foo()`
- Cannot point to the result of a call to `bar()`

Copy vs. Swap

```cpp
#include <vector>

class A {
public:
    A() = default;
    A(const A& other) : v(other.v) {}
    void swap(A& other) { v.swap(other.v); }
private:
    std::vector<int> v;
};

A foo() { return A(); }

int main() {

    A a;

    A b(a);

    a = foo();

    a.swap(b);

    return 0;
}
```

- Copy field through *copy constructor*

- Copy field through *assignment*

- Swap field through swap ( )

Move Semantics

```
// a's original std::vector<> v created
A a(0);
assert(a.size() == 0);

{
    // b's std::vector<> v created
    A b(1000);
    assert(b.size() == 1000);
    // ...

    // swap a's field v and b's field v
    a.swap(b);
    assert(a.size() == 1000);
    assert(b.size() == 0);
}

assert(a.size() == 1000);
```

- Move data instead of copying
- Can efficiently pass data from one part of the program to another without using pointers
- Previously implemented via `swap()` type operations
- All containers in the C++ Standard Library include a `swap()` operation
- If you implement any large object, consider having a `swap()`

Move Semantics Usage

```
// a's original std::vector<> v created
A a(0);
assert(a.size() == 0);

{
    // b's std::vector<> v created
    A b(1000);
    assert(b.size() == 1000);
    // ...

    // copy a's field v and b's field v
    a = b;
    assert(a.size() == 1000);
    assert(b.size() == 1000);
}

assert(a.size() == 1000);
```

```
// a's original std::vector<> v created
A a(0);
assert(a.size() == 0);

{
    // b's std::vector<> v created
    A b(1000);
    assert(b.size() == 1000);
    // ...

    // swap a's field v and b's field v
    a.swap(b);
    assert(a.size() == 1000);
    assert(b.size() == 0);
}

assert(a.size() == 1000);
```

Implementation Problems

```
// a's original std::vector<> v created
A a(0);
assert(a.size() == 0);

{
    // b's std::vector<> v created
    A b(1000);
    assert(b.size() == 1000);
    // ...

    // swap a's field v and b's field v
    a.swap(b);
    assert(a.size() == 1000);
    assert(b.size() == 0);
}

assert(a.size() == 1000);
```

- Need to add a swap ( ) method, and it's difficult to add a method to a class that is not yours

- Need to remember to use the swap ( ) method

- Not actually a move, but a swap where something is going away

- Cannot be used with temporary objects because they are not addressable

Temporary objects are *not* lvalues

```
rvalue3full.cpp:47:20: error: non-const lvalue reference to type 'A' cannot bind to a temporary of type 'A'
            a.swap(A(1000));
                   ^~~~~~~
rvalue3full.cpp:8:18: note: passing argument to parameter 'other' here
    void swap(A& other) { v.swap(other.v); }
                 ^
1 error generated.
```

```
// a's original std::vector<> v created
A a(0);
assert(a.size() == 0);

{
    // ...

    // swap a's field v and temporary's field v
    a.swap(A(1000));
    assert(a.size() == 1000);

}

assert(a.size() == 1000);
```

References

Type Modifier	Description
A&	*lvalue reference*
A&&	*rvalue reference*

Current IN Parameters

```
rValue.cpp:29:5: error: call to 'f1' is ambiguous
    f1(5);
    ^~
rValue.cpp:17:6: note: candidate function
void f1(int) {}
     ^
rValue.cpp:19:6: note: candidate function
void f1(const int&) {}
     ^
rValue.cpp:32:5: error: call to 'f1' is ambiguous
    f1(n);
    ^~
rValue.cpp:17:6: note: candidate function
void f1(int) {}
     ^
rValue.cpp:19:6: note: candidate function
void f1(const int&) {}
```

```
void f1(int) {}

void f1(const int&) {}
```

Current IN Parameters + r-value

```
rValue.cpp:29:5: error: call to 'f1' is ambiguous
    f1(5);
    ^~
rValue.cpp:17:6: note: candidate function
void f1(int) {}
     ^
rValue.cpp:19:6: note: candidate function
void f1(const int&) {}
     ^
rValue.cpp:32:5: error: call to 'f1' is ambiguous
    f1(n);
    ^~
rValue.cpp:17:6: note: candidate function
void f1(int) {}
     ^
rValue.cpp:19:6: note: candidate function
void f1(const int&) {}
```

```
void f2(int n) {}

void f2(const int&) {}

void f2(int&& n) {}
```

swap( ) Implementation

```
class A {
public:
    A(int n) : v(n) {}
    A(const A& other) : v(other.v) {}
    A(A&& other) {

        v.swap(other.v);
    }
    A& operator=(const A&) = default;
    A& operator=(A&& rhs) {

        v.swap(rhs.v);

        return *this;
    }

    int size() { return v.size(); }
private:
    std::vector<int> v;
};
```

```
int main() {

    // a's original std::vector<> v created
    A a(0);
    assert(a.size() == 0);

    // move from rvalue A(1000) to a
    a = A(1000);
    assert(a.size() == 1000);

    return 0;
}
```

`std::move()`

```
auto a = std::move(b);
```

- Converts an *lvalue* into an *rvalue*

- Moves the data from the right to the left

- The argument object is left in a *"valid but unspecified state"*

std::move() Implementation

```cpp
#include <vector>
#include <utility>
#include <cassert>

class A {
public:
    A(int n) : v(n) {}
    A(const A& other) : v(other.v) {}
    A(A&& other) : v(std::move(other.v)) {}
    A& operator=(const A&) = default;
    A& operator=(A&& other) {

        v = std::move(other.v);

        return *this;
    }

    int size() { return v.size(); }
private:
    std::vector<int> v;
};
```

```cpp
int main() {

    // a's original std::vector<> v created
    A a(0);
    assert(a.size() == 0);

    // move from rvalue A(1000) to a
    a = A(1000);
    assert(a.size() == 1000);

    return 0;
}
```

Using `std::move()`

```
A a(1);
assert(a.size() == 1);

A b(500);
assert(b.size() == 500);

a = b;
assert(a.size() == 500);
assert(b.size() == 500);
```

```
A a(1);
assert(a.size() == 1);

A b(500);
assert(b.size() == 500);

a = std::move(b);
assert(a.size() == 500);
assert(b.size() == 0);
```

Producer Consumer Benchmark and std::stack::push()

```cpp
// producer
std::vector<int> originalProduct(1000);
assert(originalProduct.size() == 1000);

// transfer to stack
std::stack<std::vector<int>> stack;
stack.push(originalProduct);
assert(stack.top().size() == 1000);
assert(originalProduct.size() == 1000);

// consumer
auto consumerProduct = stack.top();
assert(consumerProduct.size() == 1000);
assert(stack.top().size() == 1000);
stack.pop();
```

```cpp
// producer
std::vector<int> originalProduct(1000);
assert(originalProduct.size() == 1000);

// transfer to stack
std::stack<std::vector<int>> stack;
stack.push(std::move(originalProduct));
assert(stack.top().size() == 1000);
assert(originalProduct.size() == 0);

// consumer
auto consumerProduct = std::move(stack.top());
assert(consumerProduct.size() == 1000);
assert(stack.top().size() == 0);
stack.pop();
```