Intermediate Level C++

C++11: Move Semantics and rvlaue References

Yong Zhang

Objectives

In this chapter you will learn:

- Move Semantics Introduction
- lvalue and rvalue revision
- rvalue References
- Move semantics implementation
- std::move
- Perfect forwarding

Move Semantics

- A new feature introduced in C++ which is an addition to copying via copy constructor or copy assignment operator.
- Classes can have move constructor and move assignment operators.

```
class JetPlane
{
public:
    JetPlane();

    JetPlane(const JetPlane&);
    JetPlane& operator=(const JetPlane&);

    JetPlane(JetPlane&&);
    JetPlane& operator=(JetPlane&&);
};
```

Move Semantics

- **Benefits of using move semantics:**
  - Better performance
  - Better code
  - Better support for exclusive resource ownership
- **How to implement move semantics:**
  - Using rvalue references

Lvalue and Rvalue

- **Every expression in C++ is either an lvalue or an rvalue.**
- **What are lvalues and rvalues:**

lvalue

- ✓ Has a name
- ✓ Can have address taken

```
var
*ptr
arr[n]
++x
```

rvalue

- ✓ Doesn't have a name
- ✓ Can't have address taken
- ✓ this pointer

```
a * b
x++
string("abc")
```

Lvalue and Rvalue (continued)

- **The old definitions of lvalue and rvalue are not accurate:**
  - Lvalues: An expression that may appear on the left or on the right hand side of an assignment.
  - Rvalues: An expression that can only appear on the right hand side of an assignment.

```
void func(const int * pi, const int & ri) {

*pi=7;//compilation error, *pi is const

ri=8; //compilation error, ri is const

}
```

```
7==x;
```

Lvalue and Rvalue (continued)

- lvalue or rvalue based on return type for function or operator calls:

```
vector<int> v;
v[0];           // lvalue
v.size();       // rvalue

string s;
s + "abc";      // rvalue
```

Lvalue and Rvalue (continued)

- **Both lvalues and rvalues can have a const attribute, which results in four types of expressions:**

```
string f() { return string("F"); }
const string g() { return string("G"); }
JetPlane jet;
const int max_power_level = 100;

jet;                    // lvalue
max_power_level;        // const lvalue
f();                    // rvalue
g();                    // const rvalue


cout << jet.model().append("_RR").size() << endl;
```

Lvalue and Rvalue (continued)

- **Initializing references using lvalues and rvalues:**

```
JetPlane jet;
JetPlane& jet_ref = jet;

const JetPlane grounded_jet;
JetPlane& jet_ref2 = grounded_jet;      // doesn't compile

JetPlane& jet_ref3 = JetPlane();        // doesn't compile

auto make_const_jet = []() -> const JetPlane { return JetPlane(); };
JetPlane& jet_ref4 = make_const_jet(); // doesn't compile
```

```
const JetPlane grounded_jet;
const JetPlane& jet_ref2 = grounded_jet;      // OK

const JetPlane& jet_ref3 = JetPlane();        // OK

auto make_const_jet = []() -> const JetPlane { return JetPlane(); };
const JetPlane& jet_ref4 = make_const_jet(); // OK



const JetPlane& jet_ref4 = make_const_jet(); // OK

jet_ref4; // the expression jet_ref4 is an lvalue - it has a name!
```

Lvalue and Rvalue (continued)

- **Alternate C++ definition for lvalues and rvalues:**
  - Lvalues: An expression that refers to a memory location and allows us to take the address of that memory location via the & operator.
  - Rvalues: An expression that is not an lvalue, or refers to a temporary object.

```
// lvalues:
//
int i = 42;
i = 43; // ok, i is an lvalue
int* p = &i; // ok, i is an lvalue
int& foo();
foo() = 42; // ok, foo() is an lvalue
int* p1 = &foo(); // ok, foo() is an lvalue

// rvalues:
//
int foobar();
int j = 0;
j = foobar(); // ok, foobar() is an rvalue
int* p2 = &foobar(); // error, cannot take
j = 42; // ok, 42 is an rvalue
```

Rvalue References

- **Rvalue References: if X is any type, then X&& is called an rvalue reference to X.**
- **Lvalue and rvalue references are distinct types, but they behave similarly.**
- **The difference in initialization is that an rvalue reference can only be initialized with a non-const rvalue:**

```
JetPlane&& jet_ref9 = JetPlane();          // OK

JetPlane jet;
JetPlane&& jet_ref10 = jet;                // doesn't compile


const JetPlane grounded_jet;
JetPlane&& jet_ref11 = grounded_jet;       // doesn't compile

JetPlane&& jet_ref12 = make_const_jet(); // doesn't compile
```

Rvalue References (continued)

- **In C++11, there are four reference-based overloads instead of 2.**

```
void f(JetPlane& plane);
void f(const JetPlane& plane);
void f(JetPlane&& plane);
void f(const JetPlane&& plane);
```

- **When it comes to function overload resolution, lvalues prefer old-style lvalue references, whereas rvalues prefer the new rvalue references:**

```
void foo(X& x); // lvalue reference overload
void foo(X&& x); // rvalue reference overload

X x;
X foobar();

foo(x); // argument is lvalue: calls foo(X&)
foo(foobar()); // argument is rvalue: calls foo(X&&)
```

Rvalue References (continued)

- **More on overload resolution:**

```
JetPlane jet;
f(jet);                    // f(JetPlane&)

const JetPlane grounded_jet;
f(grounded_jet);           // f(const JetPlane&)

f(JetPlane());             // f(JetPlane&&)

auto make_const_jet = []() -> const JetPlane { return JetPlane(); };
f(make_const_jet());       // f(const JetPlane&&)
```

f(make_const_jet()) ⇨ f(const JetPlane&)

Move Semantics Implementation

- **Rvalue references allow a function to branch at compile time via overload resolution.**
- **This kind of overload should occur only for copy constructors and assignment operators, for the purpose of achieving move semantics.**

```
X& X::operator=(X const & rhs); // classical implementation
X& X::operator=(X&& rhs)
{
  // Move semantics: exchange content between this and rhs
  return *this;
}
```

Move Semantics Implementation (continued)

```
A(const A& rhs);
A(A&& rhs);
```

```
struct A
{
    A()
    {
        cout << "A's constructor" << endl;
    }
    A(const A& rhs)
    {
        cout << "A's copy constructor" << endl;
    }
};

vector<A> v;
cout << "==> push_back A():" << endl;
v.push_back(A());
cout << "==> push_back A():" << endl;
v.push_back(A());
```

```
==> push_back A():
A's constructor
A's copy constructor
==> push_back A():
A's constructor
A's copy constructor
A's copy constructor
```

Move Semantics Implementation (continued)

```
A(A&& rhs)
{
    cout << "A's move constructor" << endl;
}
```

```
vector<A> v;
cout << "==> push_back A():" << endl;
v.push_back(A());
cout << "==> push_back A():" << endl;
v.push_back(A());
```

```
==> push_back A():
A's constructor
A's move constructor
==> push_back A():
A's constructor
A's move constructor
A's move constructor
```

Move Semantics Implementation (continued)

- **Another example of why move semantics is better:**

```
X& X::operator=(X const & rhs)
{
  // [...]
  // Destruct the resource that m_pResource refers to.
  // Make a clone of what rhs.m_pResource refers to, and
  // attach it to m_pResource.
  // [...]
}

X foo();
X x;
// perhaps use x in various ways
x = foo();

X& X::operator=(<mystery type> rhs)
{
  // [...]
  // swap this->m_pResource and rhs.m_pResource
  // [...]
}
```

Move Semantics Implementation (continued)

```
class A
{
    double _d;
    int* _p;
    string _str;
public:
    A(A&& rhs) : _d(rhs._d), _p(rhs._p), _str(move(rhs._str))
    {
        rhs._p = nullptr;
        rhs._str.clear();
    }
    A& operator=(A&& rhs)
    {
        delete _p;

        _d = rhs._d;
        _p = rhs._p;
        _str = move(rhs._str); // careful!
        rhs._p = nullptr;
        rhs._str.clear();

        return *this;
    }
};
```

std::move

- **Tell the compile to choose a move overload when you**
  - have a named rvalue reference, or
  - an lvalue that you know isn't going to be used anymore
- **it does not move anything; it just turns an lvalue into an rvalue, so that the move constructor is invoked.**
- **Without move, the compiler would choose the copy overload.**

```
A a;
v.push_back(move(a));    // move overload used
```

Move Semantics Implementation – An Example

```
1  class ArrayWrapper
2  {
3      public:
4          ArrayWrapper (int n)
5              : _p_vals( new int[ n ] )
6              , _size( n )
7          {}
8          // copy constructor
9          ArrayWrapper (const ArrayWrapper& other)
10             : _p_vals( new int[ other._size  ] )
11             , _size( other._size )
12         {
13             for ( int i = 0; i < _size; ++i )
14             {
15                 _p_vals[ i ] = other._p_vals[ i ];
16             }
17         }
18         ~ArrayWrapper ()
19         {
20             delete [] _p_vals;
21         }
22     private:
23     int *_p_vals;
24     int _size;
25 };
```

Move Semantics Implementation – An Example (continued)

```
// move constructor
ArrayWrapper (ArrayWrapper&& other)
    : _p_vals( other._p_vals  )
    , _size( other._size )
{
    other._p_vals = NULL;
    other._size = 0;
}
```

Move Semantics Implementation – An Example (continued)

```
1  #include <utilityh> // for std::move
2
3      // move constructor
4      ArrayWrapper (ArrayWrapper&& other)
5          : _p_vals( other._p_vals  )
6          , _metadata( std::move( other._metadata ) )
7      {
8          other._p_vals = NULL;
9      }
```

- **Demo: RvalueReferences**

Perfect Forwarding

- **The problem:**

```
auto one_func(T arg1, T arg2)
{
    another_func(arg1, arg2);
}
```

  - lengthy code:

```
unique_ptr<vector<Point>> p_points(new vector<Point>(10));
```

  - Forward argument by value:

```
template<typename T, typename Arg>
unique_ptr<T> make_unique(Arg arg)
{
    return unique_ptr<T>(new T(arg));
}
```

  - Forward argument by reference:

```
template<typename T, typename Arg>
unique_ptr<T> make_unique(Arg& arg)
{
    return unique_ptr<T>(new T(arg));
}
```

```
make_unique<vector<int>>(10);  // can't convert argument from int to int&
```

Perfect Forwarding

- **Is this the solution?**

```
template<typename T, typename Arg>
unique_ptr<T> make_unique(const Arg& arg)
{
    return unique_ptr<T>(new T(arg));
}

template<typename T, typename Arg>
unique_ptr<T> make_unique(Arg& arg)
{
    return unique_ptr<T>(new T(arg));
}

int a = 10;
make_unique<vector<int>>(a);   // OK, Arg& overload
make_unique<vector<int>>(10);  // OK, const Arg& overload
```

Perfect Forwarding

- **The solution:**

```
template <typename T, typename T1, typename T2>
unique_ptr<T> make_unique(T1&& arg1, T2&& arg2)
{
    return unique_ptr<T>(new T(forward<T1>(arg1), forward<T2>(arg2)));
}
```

```
template<typename T, typename... Args>
unique_ptr<T> make_unique(Args&&... args)
{
    return unique_ptr<T>(new T(forward<Args>(args)...));
}
```

```
#include <utility>
```