Move Semantics

C++11 enables a technique called move semantics. The STL classes now have copy constructors, move constructors, copy assignment operators, and move assignment operators. This articles first examine the value categories, then will go over the basics of using move semantics.

Values Categories

Each C++ **expression** (an operator with its operands, a literal, a variable name, etc. Note: value categories do not refer to variable categories, but refer to expression categories. [JL]) is characterized by two independent properties: a **type** and a **value category**. In practice we usually only concern two types, **rvalue** and **lvalue**.

Full legal descriptions of value category: link

lvalue

An lvalue is an expression, such as a variable name or a dereferenced pointer, that represents data for which the program **can obtain an address**. Originally, an lvalue was one that could appear on the left side of an assignment statement.

```
// Lvalues can appear on the left side of
// the built-in assignment operator:
a = 0;
// The address of lvalues can be taken:
int* a_ptr = &a;
// Lvalues can bind to lvalue references:
int& a_ref = a;
// Function call returning lvalue reference:
int& bar() { static int i = 0; return i; }
&bar();
bar() = 5;
```

rvalue

Rvalue reference, indicated by using &&, can bind to rvalues—that is, values that can appear on the right-hand side of an assignment expression but for which one cannot apply the address operator.

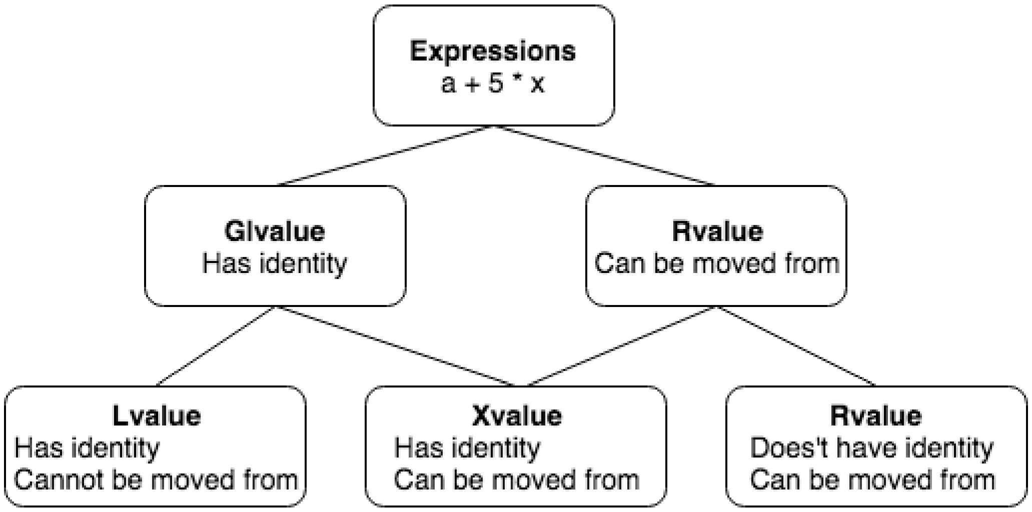
Example of rvalues:

```
int x = 10;
int y = 23;
// Rvalues can NOT appear on the left side of
// the built-in assignment operator:
5 = 0;
bar() = 0;
// The address of rvalues can NOT be taken:
&5;
&bar();
// Rvalues do NOT bind to lvalue references:
int& lv_ref0 = 5;
int& lv_ref1 = bar();
// Rvalues bind to rvalue references. Interestingly, binding an rvalue to an rvalue
// reference results in the value being stored in a location whose address can be
// taken.That is, although you can't apply the & operator to 13, you can apply it to r1.
int && r1 = 13;
```

```
// r2 really binds to is the value to which x + y evaluates at that time.
// That is, r2 binds to the value 23, and r2 is unaffected by subsequent changes to x or y.
int && r2 = x + y;
// Function returning non-reference value
int bar() { return 5; }
double && r3 = bar();
```

The Fuller Picture in C++11

The following is a very brief summary of value categorization in c++11. The graph may help you when we read std container library documents in the future.



With the introduction of move semantics in C++11, value categories were redefined to characterize two independent properties of expressions:

1. **Has identity**: it's possible to determine whether the expression refers to the same entity as another expression, such as by comparing addresses of the objects or the functions they identify (obtained directly or indirectly);
2. **Can be moved from**: move constructor, move assignment operator, or another function overload that implements move semantics can bind to the expression.

In C++11, expressions that:

- Have identity and cannot be moved from are called **lvalue** expressions.
- Have identity and can be moved from are called **xvalue** expressions.
- Do not have identity and can be moved from are called **prvalue** ("pure rvalue") expressions.
- Do not have identity and cannot be moved from are not used

The expressions that have identity are called "glvalue expressions" (glvalue stands for "generalized lvalue"). Both lvalues and xvalues are glvalue expressions. The expressions that can be moved from are called "rvalue expressions". Both prvalues and xvalues are rvalue.

Move Semantics and the Rvalue Reference

Let’s look into the copying process as it worked prior to C++11. Suppose we start with something like this:

```
vector<string> vstr;
// build up a vector of 20,000 strings, each of 1000 characters ...
vector<string> vstr_copy1(vstr); // make vstr_copy1 a copy of vstr
```

20,000,000 characters will be copied from the memory controlled by vstr to the memory controlled by vstr_copy1. Lots of work! If vstr is not needed ever after being copied to vstr_copy1, wouldn’t it be better if the compiler could just transfer ownership of the data to vstr_copy2?

Since c++11, move constructors are introduced:

```
vector( const vector& other );
vector( vector&& other );
```

A Move Examples

The following is example (shortten) given in the book of C++ primer plus. Notice how the move constructor nullified memory controlled by the passed-in parameter f. The same considerations that make move semantics appropriate for constructors make them appropriate for assignment.

Also notice the move operator parameter is not a const reference because the method alters the source object.

```
// uselessm.cpp -- an otherwise useless class with move semantics
#include <iostream>
#include <utility>
using namespace std;

// interface
class Useless
{
private:
    int n;          // number of elements
    char * pc;      // pointer to data
    static int ct;  // number of objects
    void ShowObject() const;
public:
    Useless();
    explicit Useless(int k);
    Useless(int k, char ch);
    Useless(const Useless & f); // regular copy constructor
    Useless(Useless && f);      // move constructor
    ~Useless();
    Useless operator+(const Useless & f)const;
    // need operator=() in copy and move versions
    void ShowData() const;
};

// implementation
int Useless::ct = 0;

Useless::Useless()
{
    ++ct;
    n = 0;
    pc = nullptr;
    cout << "default constructor called; number of objects: " << ct << endl;
    ShowObject();
}

Useless::Useless(int k) : n(k)
{
    ++ct;
    cout << "int constructor called; number of objects: " << ct << endl;
    pc = new char[n];
    ShowObject();
}

Useless::Useless(int k, char ch) : n(k)
{
    ++ct;
    cout << "int, char constructor called; number of objects: " << ct << endl;
    pc = new char[n];
    for (int i = 0; i < n; i++)
        pc[i] = ch;
    ShowObject();
}

Useless::Useless(const Useless & f): n(f.n)
{
    ++ct;
    cout << "copy const called; number of objects: " << ct << endl;
    pc = new char[n];
    for (int i = 0; i < n; i++)
        pc[i] = f.pc[i];
    ShowObject();
}

Useless::Useless(Useless && f): n(f.n)
{
    ++ct;
    cout << "move constructor called; number of objects: " << ct << endl;
    pc = f.pc;       // steal address
    f.pc = nullptr;  // give old object nothing in return
    f.n = 0;
    ShowObject();
}

Useless::~Useless()
{
    cout << "destructor called; objects left: " << --ct << endl;
    cout << "deleted object:\n";
    ShowObject();
    delete [] pc;
}

Useless Useless::operator+(const Useless & f)const
{
    cout << "Entering operator+()\n";
    cout << "temp object:\n";
    Useless temp = Useless(n + f.n);
    for (int i = 0; i < n; i++)
        temp.pc[i] = pc[i];
    for (int i = n; i < temp.n; i++)
        temp.pc[i] = f.pc[i - n];
    cout << "Leaving operator+()\n";
    return temp;
}

void Useless::ShowObject() const
{
    cout << "Number of elements: " << n;
    cout << " Data address: " << (void * ) pc << "\n" << endl;
}
```

```
void Useless::ShowData() const
{
    for (int i = 0; i < n; i++)
        cout << pc[i];
    cout << endl;
}

// application
int main()
{
    {
        Useless one(10, 'x');
        Useless two = one; // calls copy constructor
        Useless three(20, 'o');
        Useless four (std::move(three)); // move constructor called
        Useless five (one + three); // calls operator+(), RVO applied in Apple compiler

    }
    cin.get();
}
```

Forcing a Move

What if you want to move a lvalue? C++11 provides a simpler way to do this—use the **std::move()** function, which is declared in the utility header file. `std::move()` is equivalent to the following:

```
static_cast<typename std::remove_reference<T>::type&&>(t)
```

See it in action:

```
int main() {
  Useless one(10, 'x');
  Useless two = one +one; // calls move constructor
  Useless three, four;
  three = one; // automatic copy assignment
  four = one + two; // automatic move assignment
  four = std::move(one); // forced move assignment
}
```

One very important thing to understand is that `std::move` doesn't actually "move" anything: it simply casts an expression to an rvalue.

```
void noop_example()
{
    std::vector<int> v0{1, 2, 3, 4, 5};
    std::move(v0); // No-op.
    v0.size(); // Perfectly safe.
}
```

We should realize that the std::move() function doesn’t necessarily produce a move operation. Suppose, for instance, that Chunk is a class with private data and that we have the following code:

```
Chunk one;
...
Chunk two;
two = std::move(one); // move semantics?
```

If the Chunk class doesn't define a move assignment operator, the compiler will use the copy assignment operator. And if that also isn't defined, then assignment isn't allowed at all.

Binding & Function Overloading

An rvalue can bind to both const lvalue reference and rvalue reference. Suppose we have overloaded function as follows. Which foo() will be called?

```
void foo(int && x) {cout << "rvalue reference foo called\n";}
void foo(const int & x) {cout << "const lvalue reference foo called\n";}
int main() {
    foo(std::move(x)); // 2nd foo()
    foo(5); // 2nd foo()
}
```

More formally:

- An rvalue may be used to initialize a **const lvalue reference**, in which case the lifetime of the object identified by the rvalue is extended until the scope of the reference ends.
- An rvalue may be used to initialize an **rvalue reference**, in which case the lifetime of the object identified by the rvalue is extended until the scope of the reference ends.
- When used as a function argument and when two overloads of the function are available, one taking rvalue reference parameter and the other taking lvalue reference to const parameter, an rvalue binds to the rvalue reference overload (thus, if both copy and move constructors are available, an rvalue argument invokes the move constructor, and likewise with copy and move assignment operators).

Move Semantics & Return Statement

In the following code snippet, the std::move on tmp is unnecessary and can actually be a performance pessimization as it will inhibit return value optimization.

```
std::vector<int> return_vector(void)
{
    std::vector<int> tmp {1,2,3,4,5};
    return std::move(tmp);
}
std::vector<int> &&rval_ref = return_vector();
```

Best Practice:

```
std::vector<int> return_vector(void)
{
    std::vector<int> tmp {1,2,3,4,5};
    return tmp;
}
std::vector<int> rval_ref = return_vector();
```

There is one case returning rvalue reference makes sense. That is, if you want to return a struct/class member variable, RVO is not performed(stackoverflow reference).

```
struct Beta {
  Beta_ab ab;
  Beta_ab const& getAB() const& { return ab; }
  Beta_ab && getAB() && { return move(ab); }
  // second "&&" above is member function ref-qualifiers, also added in c++11!
  // Note that move in this case is not optional, because ab is
  // neither a local automatic nor a temporary rvalue.
};
Beta_ab ab = Beta().getAB(); // invoke move
```

Pass-by-Value and Move Idiom

What's the problem? Consider the following constructor for class Person.

```
struct Person
{
    std::string name;
    Person(const std::string& name) : name{name} {} // 1 Copy
    Person(std::string&& name) : name{std::move(name)} {} // 1 Move
};
```

The code is optimal for users of the class. However, if the number of parameters are N, we need to write 2^N overloaded constructor. Ouch! That's when the "pass-by-value and move" comes in handy.

```
struct Person
{
    std::string name;
    // * Lvalue => 1 copy + 1 move
    // * Rvalue => 2 moves
    // Move is cheap; close to optimal
    Person(std::string name) : name{std::move(name)} {}
};
```

Standard library support of moveble type

Many, if not all, of std containers are move-aware.

```
// std::vector::push_back
void push_back( const T& value );
void push_back( T&& value );

// std::vector::operator=
vector& operator=( const vector& other );
vector& operator=( vector&& other );

// std::vector::vector
```

```
vector( const vector& other );
vector( vector&& other );
```

The entire container can be moved to a destination, or items can be moved inside the containers.

```
std::vector<int> get_vector(int x);
void consume_vector(std::map<int, std::vector<int>> m);

std::map<int, std::vector<int>> map_of_vector;
for(int i = 0; i < 100; ++i)
{
    // invoke vector move assignment operator
    map_of_vector[i] = get_vector(i);
}
// invoke map move constructor operator
consume_vector(std::move(map_of_vector));
```

Some classes provided by the Standard Library can only be moved.

```
std::thread t{[] { std::cout << "hello!"; }};
// auto t_copy = t; // Does not compile.
auto t_move = std::move(t);

std::unique_ptr<int> up = std::make_unique<int>(1);
// auto up_copy = up; // Does not compile.
auto up_move = std::move(up);
```

Containers allow us to go one step further, and sometimes allow us to entirely avoid the move. This can happen when an item is being constructed "in place" inside the container. This operation is called "emplacement" and is supported by most Standard Library containers.

```
// std::vector::emplace_back declaration
template< class... Args >
void emplace_back( Args&&... args );

struct bar
{
    int x;
    bar(int x) : x{x} { std::cout << "bar(int)\n"; }
    bar(const bar&)    { std::cout << "bar(const bar&)\n"; }
    bar(bar&&)         { std::cout << "bar(bar&&)\n"; }
    bar()              { std::cout << "bar()\n"; }
};

void vector_emplacement()
{
    std::vector<bar> v;
    v.reserve(10);

    // Moves `bar` temporary
    v.push_back(bar{42});

    // Constructs `bar` instance "in place"
    v.emplace_back(42);
}
```