RValue References

an RValue is an unnamed value which typically appears on the right side of an assignment. This is in contrast to an LValue, which appears on the left hand side of an assignment.

Every expression in C++ is either an Lvalue or an Rvalue. Rvalues are temporary values which only exist for the duration of one expression. Lvalues persist beyond the boundary of a single expression.

lvalues:

- have names
- can take its address

rvalues

- dont have names
- cannot have its address taken
- this pointer is an rvalue fyi

Const

Of course, you also have to consider the constness of a variablle

Overload Resolution

- maintain const correctness. wont bind const value to a non const ref
- lvalues are always bound to lvalue references; bind rvalues to rvalue references if possible
- if rule 2 isnt enough to resolve ambiguity, chose an overload which preserves const-ness

Example

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

```
Jet jet;
f(jet)
```

chose `f(Jet&)`

```
const Jet grounded_jet
f(grounded_jet)
```

chose `f(const Jet&)`

`f(Jet())`

chose `f(Jet&&)`

`auto make_const_jet = []() -> const Jet {   return Jet(); }`

chose `f(const Jet&&)`

Important - const correctness

If however, the overload taking a const lvalue did not exist (ie no `f(const Jet&&)`),
then

`auto make_const_jet = []() -> const Jet {   return Jet(); }`

would bind to `f(const Jet&)` and not `f(Jet&&)`.

Move Semantics

- Better Performance
- More clarity of intention of code
- Better support for exclusive resource ownership

You should implement move constructors and move operators whenever you can.

Compiler Generated Move Operations

The compiler will generate move operations in certain circumstances:

- class does not have user declared copy or assignment operator
- class does not have user declared move assignment operator
- class does not have user declared destructor
- class does not implicitly mark move constructor as deleted

Implementing move semantics in your class

Move operations need to do the following:

- get rid of the class instance's current state
- transfer ownership
- leave original instance in valid state

Example class fragment

```
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_; // delete existing memory
       d_ = rhs.d_;
       p_ = rhs.p_;
       str_ = move(rhs.str_);
       // clean up
       rhs.p_ = nullptr;
       rhs.str_.clear();

       return *this;
   }
};
```

- for builtin types, set the source member to the target member (eg d_ above)
- if the source member is a pointer, also set the source member to nullptr
- non-builtin rvalues - use std::move to invoke their move constructors

Other Move variants

You are not limited to the move constructor and move assignment operator when implementing move overloads. You can do this for your setters as well.

```
void A::set_name(const string& nm) {
    str_ = nm;
}

void A::set_name(string&& nm) {
    str_ = move(nm);
    nm.clear();
}
```

What is std::move ?

The move function is used to tell the compiler to chose a move overload when you have a named rvalue reference or when you have an lvalue that you know will not be used anymore. Without std::move, the compiler would chose the copy overload.
No lvalue is special. You can use the move function on any of them.

```
vector<A> v;
{
    A a;
    v.push_back(move(a));
}
```

So all move does is convert an lvalue into an rvalue.

You can also use `static_cast<T&&>` instead, but move is shorter and clearer.

Guidelines

Don't return const objects from functions

While returning const values from functions are legal, they are not useful in the context of move semantics. Because of the new overload resolution rules, returning a const value from a function - which was useless but harmless before - now can result in a copy operation being selected instead of a move operation, as a const rvalue cannot bind to a non-const rvalue reference.

```
auto make_const_jet = []() -> const Jet { return Jet(); };
Jet jet(make_const_jet()); // copy constructor called instead of move constructor
```

Don't declare const rvalue reference parameters in functions

You wont be able to move from them either because of overload rules.

- `f(const T&&)` // BAD

Derived Class Construction

If you have a class with move semantics which you wish to derive from, you need to be careful in the derived class.

```
// WHOOPS, this calls the COPY constructor
Derived(Derived&& rhs) : Base(rhs) {}
```

Instead, do this:

```
Derived(Derived&& rhs) : Base(move(rhs)) {}
```

Move constructor in terms of assignment

```
B(B&& rhs) {
    // wrong - invokes copy assignment operator
    *this = rhs;
}
```

The above is wrong.

```
B(B&& rhs) {
    *this = move(rhs);
}
```

The above is correct.

Self Assignment

```
B& operator=(B&& rhs) {
   p_ = rhs.p_;
   rhs.p_ = nullptr;
   return *this;
}

B b;
b = move(b);
*b.p_; // undefined behavior. p_ is null;
```

Always add the check to the assignment move operator:

```
B& operator=(B&& rhs) {
    if(this == &rhs) return *this;

     p_ = rhs.p_;
       rhs.p_ = nullptr;
       return *this;
}
```

Do NOT declare move constructor explicit

This will result in a copy constructor being chosen instead.

NO

```
explicit A(A&&) // BAD
```

Reference Qualifiers for Member Functions

So far, we have seen overloads based on parameter signatures, however you can also do the following:

```
struct A {
    bool run() const & { return true; };
    bool run() && { return false; };
};

A a;
a.run(); // calls run() const &
A().run(); // calls run() &&
```

With this feature you can optimize. For instance, with the plus operator:

```
Counter operator+(const Counter& rhs) const & {
    Counter c(*this); // take copy
    c += other;
    return c;
}
```

The second form here will get used if Counter is used, for example, with temporary objects

```
Counter operator+(const Counter& other) && {
    *this += other;
    return move(*this);
}
```

Note: Do not mix reference and non reference qualified overloads of the same function

```
struct A {
    bool run() const { return false; }
    bool run() && { return true; } // error
};
```

Move Only Types

Move semantics can also help express semantics of ownership. For example, the unique_ptr from the STL. Threads, locks, and streams are other examples.

You create a move only type by implementing move constructors and move assignment operators but not copy constructors or assignment operators. You should explicitly mark the copy and assignment operators as deleted.

```
MoveOnly(const MoveOnly& rhs) = delete;
MoveOnly& operator=(const MoveOnly& rhs) = delete;
```