Move Semantics, Perfect Forwarding, constexpr

C++11 Features in GCC 4.8

pluralsight
hardcore developer training

Overview

- Move semantics
- Rvalue references
- Perfect forwarding
- constexpr

Move Semantics

```
vector<string> v;

v.push_back(string("a"));
v.push_back(string("b"));



string s = string("Boeing") + "737" + "-" + "300";
```

Move Semantics

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

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

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

What Are the Benefits?

- Better performance

- More clarity of intention in the code

```
Surface3D get_surface(const Latitude& lat, const Longitude& lon)
{
    Surface3D surface;
    // load up millions of points making up the surface
    return surface;
}
```

- Better support for exclusive resource ownership

How Does This Stuff Work?

References

Lvalue

Rvalue

Revision Part 1: lvalue vs. rvalue

- Attribute of expressions, not variables
- l and r don't stand for anything in particular

lvalue	rvalue
✓ Has a name	✓ Doesn't have a name
✓ Can have address taken	✓ Can't have address taken
	✓ this pointer

lvalue examples:

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

rvalue examples:

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

Revision Part 1: lvalue vs. rvalue

```
vector<int> v;
v[0];           // lvalue because vector<int>::operator[] returns int&
v.size();       // rvalue because because vector<int>::size() returns size_t

string s;
s + "abc";      // rvalue because string::operator+ returns string
```

Revision Part 2: const + lvalue/rvalue

```
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;  // append modifies the string
```

Revision Part 3: Reference Initialization

```
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
```

Revision Part 3: Reference Initialization

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

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!
```

rvalue References

JetPlane&& rvalue_ref

References

Lvalue (T&) Rvalue (T&&)

rvalue References

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

JetPlane jet;
JetPlane&& jet_ref10 = jet;               // doesn't compile to prevent accidental
                                          // modification of an lvalue

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

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

Overload Resolution

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

- **Maintain const-correctness - don’t bind const value to non-const ref**
- **Bind lvalues to lvalue refs; bind rvalues to rvalue refs if possible**
- **If rule 2 isn’t enough to resolve ambiguity, choose an overload which preserves const-ness**

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

```
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

```
A(A&& rhs) noexcept
{
    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
```

Compiler Generated Move Operations

- No user-declared copy constructor or copy assignment operator
- No user-declared move assignment operator
- No user-declared destructor
- The move constructor wouldn't be implicitly marked as deleted

Implementing Your Own Move Operations

```
class A
{
    double _d;
    int* _p;
    string _str;
};
```

Implementing Your Own Move Operations

```
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;
    }
};
```

Implementing Your Own Move Operations

```
void JetPlane::set_model(const string& model)
{
    _model = model;
}

void JetPlane::set_model(string&& model)
{
    _model = move(model); // careful: model is a named rvalue ref so it's an
                          // lvalue; use std::move to force a move operation
    model.clear();
}
```

```
string model("Airbus 320");
JetPlane jet;

jet.set_model(model);                  // copy overload used
jet.set_model(string("Airbus 320"));   // move overload used
```

std::move

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

move <=> static_cast<T&&>

rvalue References to const Values

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

JetPlane jet(make_const_jet());    // copy constructor invoked
```

const T f()

f(const T&&)

Derived Class Construction

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

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

Move Construction in Terms of Assignment

```
B(B&& rhs)
{
    *this = rhs; // WRONG: invokes the copy assignment operator
}
```

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

Self Assignment

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

```
if (this == &rhs)
    return *this;
```

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

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

Explicit Move Constructors

```
explicit A(A&&)
```

Reference Qualifiers for Member Functions

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

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

```
void f(const T&);
void f(T&&);
```

Reference Qualifiers for Member Functions

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

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

Reference Qualifiers for Member Functions

```
struct Curious
{
    int _count = 10;
    Curious& operator ++() { ++_count; return *this; }
    Curious* operator &() { return this; }
};
```

```
Curious() = Curious();      // assign to rvalue
Curious& c = ++Curious();   // c is a dangling reference
&Curious();                 // address of rvalue
```

```
struct Curious
{
    int _count = 10;
    Curious& operator ++() & { ++_count; return *this; }
    Curious* operator &() & { return this; }
};
```

Reference Qualifiers for Member Functions

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

Move-only Types

```
class MoveOnly
{
    int* _p;
public:
    MoveOnly() : _p(new int(10)) {}
    ~MoveOnly() { delete _p; }

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

    MoveOnly(MoveOnly&& rhs)
    {
        *this = move(rhs);
    }

    MoveOnly& operator=(MoveOnly&& rhs)
    {
        if (this == &rhs)
            return *this;
        _p = rhs._p;
        rhs._p = nullptr;
        return *this;
    }
};
```

Move-only Types

```
MoveOnly a;

MoveOnly b(a);          // copying, doesn't compile

MoveOnly c(move(a));    // OK

MoveOnly d;
d = move(b);            // OK
```

Perfect Forwarding Problem and Solution

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

```
template<typename T, typename Arg>
unique_ptr<T> make_unique(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));
}
```

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

Perfect Forwarding Problem and 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 Problem and 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>
```

Reference Collapsing and rvalues in Templates

```
Point p1(10, 10);
using PointRef = Point&;
PointRef& p2 = p1;
```

A& &	*becomes*	A&
A& &&	*becomes*	A&
A&& &	*becomes*	A&
A&& &&	*becomes*	A&&

Reference Collapsing and rvalues in Templates

```
template<typename T> void f(T&&);
```

f(lvalue A) => T is A& => f(A& &&) => f(A&)

f(rvalue A) => T is A => f(A&&)

How the forward Template Works

```
template<class T>
T&& forward(typename remove_reference<T>::type& arg)
{
    // forward arg, given explicitly specified type parameter
    return static_cast<T&&>(arg);
}


template<typename T>
struct remove_reference
{
    typedef T type;
};

template<typename T>
struct remove_reference<T&>
{
    typedef T type;
};

template<typename T>
struct remove_reference<T&&>
{
    typedef T type;
};
```

How the forward Template Works

```
template <typename T>
T&& forward(T&& arg)
{
    return arg;
}
```

How the forward Template Works

```
string model("Boeing 787");
auto sp = make_unique<JetPlane>(model);


unique_ptr<JetPlane> make_unique(JetPlane& && arg1)
{
    return unique_ptr<JetPlane>(new JetPlane(forward<JetPlane&>(arg1)));
}

JetPlane& && forward(remove_reference<JetPlane&>::type& arg)
{
    return (JetPlane& &&) arg;
}


unique_ptr<JetPlane> make_unique(JetPlane& arg1)
{
    return unique_ptr<JetPlane>(new JetPlane(forward<JetPlane&>(arg1)));
}

JetPlane& forward(JetPlane& arg)
{
    return (JetPlane&) arg;
}
```

How the forward Template Works

```
auto sp = make_unique<JetPlane>("Boeing 787");


unique_ptr<JetPlane> make_unique(JetPlane&& && arg1)
{
    return unique_ptr<JetPlane>(new JetPlane(forward<JetPlane>(arg1)));
}

JetPlane&& forward(remove_reference<JetPlane>::type& arg)
{
    return (JetPlane&&) arg;
}


unique_ptr<JetPlane> make_unique(JetPlane&& arg1)
{
    return unique_ptr<JetPlane>(new JetPlane(forward<JetPlane>(arg1)));
}

JetPlane&& forward(JetPlane& arg)
{
    return (JetPlane&&) arg;
}
```

The Implementation of std::move

```
template<class T>
typename remove_reference<T>::type&& move(T&& arg)
{
    return static_cast<typename remove_reference<T>::type&&>(arg);
}
```

```
int a = 10;
move(a);
```

```
remove_reference<int&>::type&& move(int& && arg)
{
    return static_cast<remove_reference<int&>::type&&>(arg);
}
```

```
int&& move(int& arg)
{
    return (int&&) arg;
}
```

The Implementation of std::move

```
remove_reference<int>::type&& move(int&& arg)
{
    return static_cast<remove_reference<int>::type&&>(arg);
}
```

```
int&& move(int&& arg)
{
    return (int&&) arg;
}
```

constexpr Mechanism

Variables

Functions

What Else Is It Good for?

- Ensure constant initialization at compile time
- Constant expressions can be used in case labels etc.
- Guaranteed not to cause race conditions

What's in Constant Expression?

Integer literal

Floating point literal

Enumerator

Address

constexpr Classes

constexpr Functions

constexpr Variables

constexpr Variables

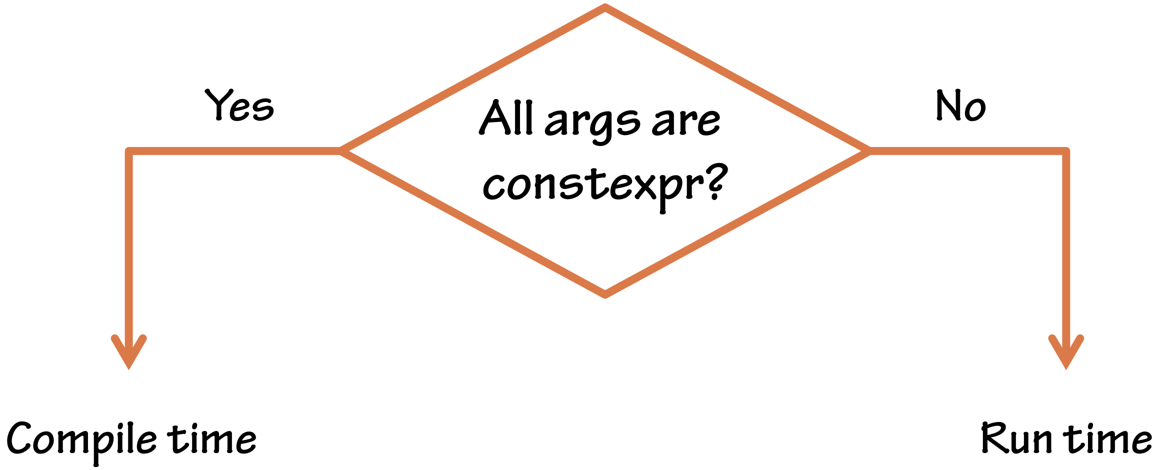
```
constexpr auto c_dimensions = 3;

constexpr auto c_threshold = 42.5;

constexpr auto c_name = "constexpr evaluator";
```

const and constexpr

```
auto a = 10;

const auto b = a;

constexpr auto d = b;
```

```
const auto c = 10;

constexpr auto e = c;
```

constexpr Functions




```
{
    return 10;
}
```

constexpr Functions

```
constexpr long fibonacci(int n)
{
    return n < 1 ? -1 :
        (n == 1 || n == 2 ? 1 : fibonacci(n - 1) + fibonacci(n - 2));
};


enum Fibonacci
{
    Ninth = fibonacci(9),
    Tenth = fibonacci(10)
};


auto a = 4, b = 6;
cout << fibonacci(a + b) << endl;  // outputs 55


template<typename T>
constexpr auto square(const T& v) ->
decltype(v * v)
{
    return v * v;
}
```

Literal Types

- **void**
- **Scalar types**
- **Reference types referring to literal types**
- **Arrays of literal types**
- **Classes with the following:**
  - Trivial destructor
  - All non-static data members and base classes are also literal types
  - It’s an aggregate type, or has at least one constexpr constructor which isn’t a copy or move constructor

Literal Types

```
class Complex
{
    double _real, _imaginary;
public:
    constexpr Complex(double real, double imaginary)
        : _real(real), _imaginary(imaginary)
    {}

    constexpr double real() const { return _real; }

    constexpr double imaginary() const { return _imaginary; }
};


constexpr Complex c1(1, 2);
```

Literal Types

```
constexpr Complex operator+(const Complex& lhs, const Complex& rhs)
{
    return Complex(lhs.real() + rhs.real(), lhs.imaginary() + rhs.imaginary());
}


constexpr Complex c1(1, 2);
constexpr Complex c2(3, 4);

constexpr Complex c3 = c1 + c2;
```

A Couple More Notes on constexpr

```
constexpr int percentage(int i)
{
    return (i >= 0 && i <= 100) ? i : throw "out of range";
}

constexpr int val = percentage(99);
```

Summary

- Move semantics and reference machinery
- Perfect forwarding
- Constant expressions and compile time evaluation