Google

Practical Move Semantics

titus@google.com

Have you ever?

- Had compilation errors with `std::unique_ptr` and the delete'd copy c'tor?
- Wondered whether you were triggering a copy or a move of a container?
- Wondered whether a call to `std::move` was necessary?

Then this talk is for you!

Start with `std::unique_ptr`

The name was well chosen.

How do we identify when a `std::unique_ptr` is no longer unique?

```
 1: std::unique_ptr<Foo> NewFoo() {
 2:   return std::make_unique<Foo>(1);
 3: }
 4:
 5: void AcceptFoo(std::unique_ptr<Foo> f) { f->PrintDebugString(); }
 6:
 7: void Simple() {
 8:   AcceptFoo(NewFoo());
 9: }
10:
11: void DoesNotBuild() {
12:   std::unique_ptr<Foo> g = NewFoo();
13:   AcceptFoo(g); // DOES NOT COMPILE!
14: }
15:
16: void SmarterThanTheCompilerButNot () {
17:  Foo* j = new Foo(2);
18:  // Compiles, BUT VIOLATES THE RULE and will double-delete at runtime.
19:  std::unique_ptr<Foo> k(j);
19:  std::unique_ptr<Foo> l(j);
20: }
```

```
 1: std::unique_ptr<Foo> NewFoo() {
 2:   return std::make_unique<Foo>(1);
 3: }
 4:
 5: void AcceptFoo(std::unique_ptr<Foo> f) { f->PrintDebugString(); }
 6:
 7: void Simple() {
 8:   AcceptFoo(NewFoo());
 9: }
10:
11: void DoesNotBuild() {
12:   std::unique_ptr<Foo> g = NewFoo();
13:   AcceptFoo(g); // DOES NOT COMPILE!
14: }
15:
16: void SmarterThanTheCompilerButNot () {
17:  Foo* j = new Foo(2);
18:  // Compiles, BUT VIOLATES THE RULE and will double-delete at runtime.
19:  std::unique_ptr<Foo> k(j);
19:  std::unique_ptr<Foo> l(j);
20: }
```

```
 1: std::unique_ptr<Foo> NewFoo() {
 2:   return std::make_unique<Foo>(1);
 3: }
 4:
 5: void AcceptFoo(std::unique_ptr<Foo> f) { f->PrintDebugString(); }
 6:
 7: void Simple() {
 8:   AcceptFoo(NewFoo());
 9: }
10:
11: void DoesNotBuild() {
12:   std::unique_ptr<Foo> g = NewFoo();
13:   AcceptFoo(g); // DOES NOT COMPILE!
14: }
15:
16: void SmarterThanTheCompilerButNot () {
17:  Foo* j = new Foo(2);
18:  // Compiles, BUT VIOLATES THE RULE and will double-delete at runtime.
19:  std::unique_ptr<Foo> k(j);
19:  std::unique_ptr<Foo> l(j);
20: }
```

```
 1: std::unique_ptr<Foo> NewFoo() {
 2:   return std::make_unique<Foo>(1);
 3: }
 4:
 5: void AcceptFoo(std::unique_ptr<Foo> f) { f->PrintDebugString(); }
 6:
 7: void Simple() {
 8:   AcceptFoo(NewFoo());
 9: }
10:
11: void EraseTheName() {
12:   std::unique_ptr<Foo> g = NewFoo();
13:   AcceptFoo(std::move(g)); // DOES COMPILE!
14: }
15:
16: void SmarterThanTheCompilerButNot() {
17:  Foo* j = new Foo(2);
18:  // Compiles, BUT VIOLATES THE RULE and will double-delete at runtime.
19:  std::unique_ptr<Foo> k(j);
19:  std::unique_ptr<Foo> l(j);
20: }
```

```
 1: std::vector<Foo> NewFoo() {
 2:   return std::vector<Foo>({1});
 3: }
 4:
 5: void AcceptFoo(std::vector<Foo> f) { … }
 6:
 7: void Simple() {
 8:   AcceptFoo(NewFoo());
 9: }
10:
11: void EraseTheName() {
12:   std::vector<Foo> g = NewFoo();
13:   AcceptFoo(std::move(g)); // DOES NOT COPY!
14: }
```

Two names: a copy

```
vector<int> foo;
FillAVectorOfIntsByOutputParameterSoNobodyThinksAboutCopies (&foo);
vector<int> bar = foo; // Yep, this is a copy.

map<int, string> my_map;
string forty_two = "42";
my_map[5] = forty_two; // Also a copy: my_map[5] counts as a name.
```

One name: a move

```
vector<int> GetSomeInts() {
  vector<int> ret = {1, 2, 3, 4};
  return ret;
}

// Just a move: either "ret" or "foo" has the data, but never both at
// once.
vector<int> foo = GetSomeInts();

// Also a move: std::move makes the old name no longer count.
vector<int> bar = std::move(foo);
```

No names: temporaries

```
void OperatesOnVector(const vector<int>& v);

// No copies: the values in the vector returned by GetSomeInts()
// will be moved (O(1)) into the temporary constructed between these
// calls and passed by reference into OperatesOnVector().
OperatesOnVector(GetSomeInts());
```

Test

```
std::vector<string> foo() {
  std::vector<string> ret;
  ret.resize(100);
  for (int i = 0; i < 100; ++i) {
    ret.push_back(std::to_string(i));
  }
  return ret;
}

void f() {
  auto vec = foo();  // O(1) or O(n)?
}
```

Test

```
std::vector<string> foo() {
  static std::vector<string> ret;
  ret.resize(100);
  for (int i = 0; i < 100; ++i) {
    ret.push_back(std::to_string(i));
  }
  return ret;
}

void f() {
  auto vec = foo();  // O(1) or O(n)?
}
```

Test

```
std::vector<string> foo() {
  std::vector<string> ret;
  ret.resize(100);
  for (int i = 0; i < 100; ++i) {
    ret.push_back(std::to_string(i));
  }
  return ret;
}

void f() {
  auto vec = std::move(foo());  // O(1) or O(n)?
}
```

Two weird notes

- Don’t touch moved-from objects
  - Without guarantees from the API provider about what is in them after a move
- `std::move` doesn’t actually do anything itself
  - It’s a cast, the move-constructor or move-assignment operator does the work

Questions?