C++ Move Semantics

Michael Gopshtein

EYEWAY
ALL IS ILLUMINATED

Why `move`

1. Avoid copying from [large] object when we no longer need the original instance.

```
vector<int> v;
// fill the vector
return v;
```

2. Transferring ownership

```
v.push_back(make_unique<A>());
```

Moving my class

```
class MyClass {
   void *_largeBuffer;

public:
   MyClass(MyClass &&from)
      : _largeBuffer(from._largeBuffer)
   {
      from._largeBuffer = nullptr;
   }

   ~MyClass() { delete _largeBuffer; }

   //...
};
```

```
MyClass m1{…};
//...
MyClass m2{ std::move(m1); }
//...
// destructors of m1 and m2
// will be eventually called
```

Original instance:

- Keep a consistent state
- Destructor will be called normally

Usage Examples

```
MyClass create();

void use() {
   MyClass mc{ create(); }
}
```

```
MyClass temp;
//...
vector<MyClass> v;
v.push_back( std::move(temp) );
```

Swapping my class

```
class MyClass {
    // ... Destructor, Copy/Move Constructors

    MyClass & operator=(MyClass &&other) {
        swap(other);
        return *this;
    }

    void swap(MyClass &other) {
        std::swap(largeBuffer, other._largeBuffer);
    }
};
```

Swapping my class

```
class MyClass {
   // ... Destructor, Copy/Move Constructors

   MyClass & operator=(MyClass &&other) {
      swap(other);
      return *this;
   }

   MyClass & operator=(const MyClass &other) {
      MyClass temp(other);
      swap(temp); ;
      return *this;
   }

   void swap(MyClass &other) {
      std::swap(largeBuffer, other._largeBuffer);
   }
};
```

How to define constructors for UseNoisy?

```
class Noisy {
    // heavy construction (by int)
    // heavy copying
    // lightweight move
};
```

```
class UseNoisy {
    Noisy _n;
};
```

```cpp
class Noisy {
   // heavy construction (by int)
   // heavy copying
   // lightweight move
};
```

```cpp
class UseNoisy {
   Noisy _n;
};
```

Use cases:

```cpp
UseNoisy un{ 1 };
```

```cpp
Noisy n;
UseNoisy un{ n };
```

```cpp
Noisy ntemp;
UseNoisy un{ std::move(ntemp) };
```

```
class Noisy {
    // heavy construction
    // heavy copying
    // lightweight move
};
```

Use cases:

```
UseNoisy un{ 1 };
```

```
Noisy n{ … };
UseNoisy un{ n };
```

```
Noisy ntemp;
UseNoisy un{ std::move(ntemp) };
```

std::move

(not just move)

```cpp
class Noisy {
   // heavy construction
   // heavy copying
   // lightweight move
};
```

```cpp
class UseNoisy {
   Noisy _n;
};
```

Use cases:

```cpp
UseNoisy un{ 1 };
```

```
1 construct
```

```cpp
Noisy n{ … };
UseNoisy un{ n };
```

```
1 copy
```

```cpp
Noisy ntemp;
UseNoisy un{ std::move(ntemp) };
```

```
1 move
```

<code>**Version 1**</code>

```
UseNoisy(int i) : _noisy(i) {}
UseNoisy(const Noisy &n) : _noisy(n) {}
UseNoisy(Noisy &&n) :_noisy(std::move(n)) {}
```

Results

Init by:	Ideal	All overloads		
`int`	1c	1c		
copy `Noisy`	1c	1c		
move `Noisy`	1m	1m		

<code>Version 2</code>

```
UseNoisy(Noisy n) : _noisy(std::move(n)) {}
```

Results

Init by:	Ideal	All overloads	Just by value	
`int`	1c	1c	1c + 1m	
copy `Noisy`	1c	1c	1c + 1m	
move `Noisy`	1m	1m	1m + 1m	

- The best version so far is very verbose.
  assume initialization with multiple parameters…
- The simplest *by value* version gives close to optimal results!

```
UseNoisy(Noisy &&noisy)
    : _noisy(noisy)
{}
```

```
UseNoisy(Noisy &&noisy)
    : _noisy(std::move(noisy))
{}
```

Lvalue

Stuff with a name and known address

```
a
a.field
```

```
a = createA();
```

Rvalue

Temp stuff without a name, can't take address of

```
createA()
A{1}
```

Lvalue

Stuff with a name and known address

```
a = createA();
```

Rvalue

Temp stuff without a name, can't take address of

`Noisy&`

Lvalue reference

(the *normal* reference)

`Noisy&&`

Rvalue reference (C++11)

Lvalue

`Noisy&`

Lvalue reference

(the *normal* reference)

Rvalue

`Noisy&&`

Rvalue reference
(C++11)

```
UseNoisy(Noisy &&noisy)
    : _noisy(noisy)
{}
```

Inside the constructor, *noisy* has a name, and becomes “lvalue”.

- Can be used later in the function, for example.

```
UseNoisy(Noisy &&noisy)
    : _noisy(std::move(noisy))
{}
```

Results

Init by:	Ideal	All overloads	Just by value	
`int`	1c	1c	1c + 1m	
copy `Noisy`	1c	1c	1c + 1m	
move `Noisy`	1m	1m	1m + 1m	

Still imperfect

Why do we love C++?

```
template<typename T>
void func(T&& t) {
}
```

Here `t` is not an *rvalue reference*!

It is called a ***universal*** *reference*

Why do we love C++?

```
template<typename T>
void func(T&& t);
```

How to tell:

Is T a “deduced parameter”?

```
A&& a = get();
```

```
template<typename T>
void func(vector<T>&& v);
```

```
auto&& a = get();
```

```
template<typename T>
void func(T&& t);
```

Perfect forwarding

```
template<typename T>
UseNoisy(T&& noisy)
    : _noisy(std::forward<T>(noisy))
{}
```

A common pattern to handle both *lvalues* and *rvalues*.

Just works ☺

emplace

```
template<typename... T>
UseNoisy(T&&... noisy)
    : _noisy(std::forward<T>(noisy)...)
{}
```

All STL containers have support for *emplace* functions, which initialize the value once in the container itself, by *forwarding* init parameters.

<code>**Version 3**</code>

```
template<typename T>
UseNoisy(T&& t) : _noisy(std::forward<T>(t)) {}
```

Results

Init by:	Ideal	All overloads	Just by value	Perfect FW
`int`	1c	1c	1c + 1m	1c
copy `Noisy`	1c	1c	1c + 1m	1c
move `Noisy`	1m	1m	1m + 1m	1m

* there's some copy constructor issue remaining in the last approach… but we'll ignore it for now

Don’t move return values

```
Noisy n = construct();
```

	Debug build	Release build

`Noisy construct() {`
`    return Noisy{1};`
`}`
	1 construct	1 construct

Return Value Optimization (RVO)

Don’t move return values

```
Noisy n = construct();
```

	Debug build	Release build

`Noisy construct() {`
`    return Noisy{1};`
`}`
	1 construct	1 construct

`Noisy construct() {`
`    Noisy result{1};`
`    return result;`
`}`
	1 construct
	1 move
		1 construct

Return Value Optimization (RVO)

Named RVO (NRVO)

Don’t move return values

```
Noisy n = construct();
```

	Debug build	Release build

`Noisy construct() {`
`   return Noisy{1};`
`}`
	1 construct	1 construct

`Noisy construct() {`
`   Noisy result{1};`
`   return result;`
`}`
	1 construct
	1 move
		1 construct

`Noisy construct() {`
`   Noisy result{1};`
`   return std::move(result);`
`}`
	1 construct
	1 move
		1 construct
		1 move

Return Value Optimization (RVO)

Named RVO (NRVO)

Auto-generated moves

Constructor and assignment are NOT generated if you defined one of:

- Copy constructor or assignment
- Destructor
- Move constructor or assignment

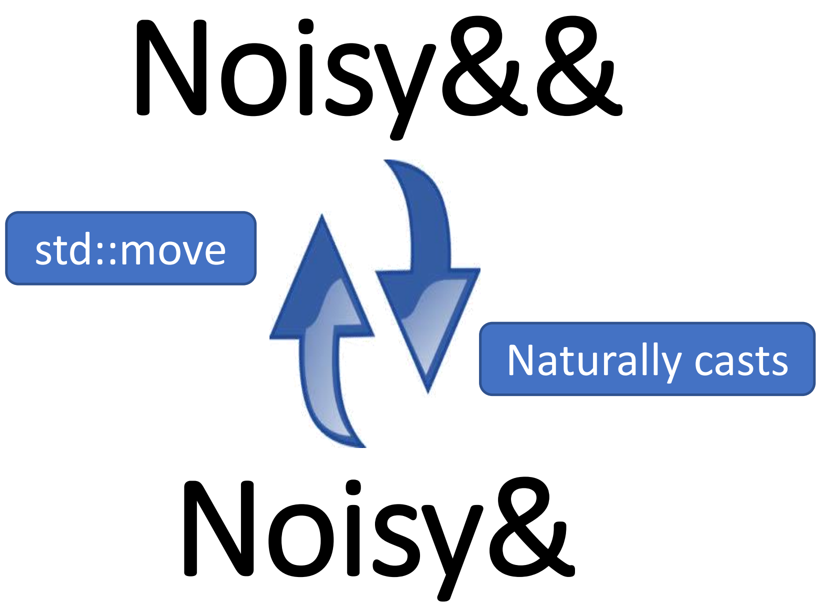
Can *force* generation:

```
class UseNoisy {
   ~UseNoisy() {}

   UseNoisy(UseNoisy&&) = default;
   UseNoisy & operator=(UseNoisy&&) = default;
};
```

```cpp
class TypeNoMove {
public:
	TypeNoMove(const TypeNoMove&) {
		cout << "TypeNoMove(copy)" << '\n';
	}

	TypeNoMove() = default;
};
```



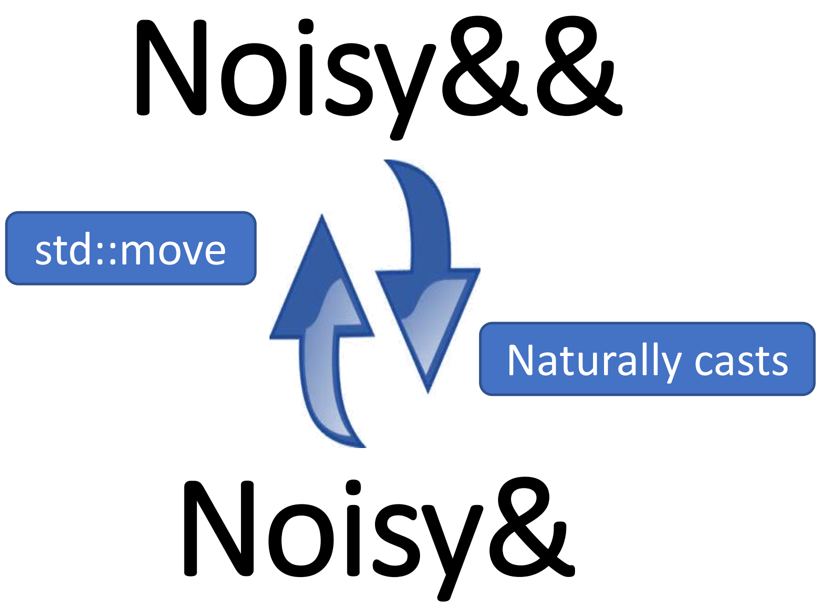

```cpp
TypeNoMove t1;
TypeNoMove t2{ std::move(t1) };  // copies!
```

```
#include <type_traits>
std::is_move_constructible<TypeNoMove> ??
```

```
class TypeNoMove {
public:
	TypeNoMove(const TypeNoMove&) {
		cout << "TypeNoMove(copy)" << '\n';
	}

	TypeNoMove() = default;

	TypeNoMove(TypeNoMove&&) = delete;
};
```




```
TypeNoMove t1;
TypeNoMove t2{ std::move(t1) }; // does not compile
```

#include <type_traits>
std::**is_move_constructible**<TypeNoMove> ??

Moving ownership

```
vector<unique_ptr<A>> vec;
```

```
vec.push_back(make_unique<A>());
```

```
auto pa = make_unique<A>();
vec.push_back(std::move(pa));
```

std::move

Use **swap** for move assignment

Don’t std::move **return** value

Pass more objects **by value**

(T&& t) becomes “lvalue” in processing context, **explicitly** std::move it

Be careful with **&&**

Default operators