Modern C++ API Design

Rvalue references and modern type design

Titus Winters (titus@google.com)

A Talk in Three Parts

A Refresher on Rvalue-References

How to use Rvalue-References in API Design

How to use those APIs in Type Design

Refresher: rvalue refs

What is an rvalue ref?

Refresher: rvalue refs

What is an rvalue ref?

A reference to an rvalue

Refresher: rvalue refs

What is an rvalue?

Refresher: rvalue refs

What is an rvalue?

Something you could only have on the right side of an assignment.

Refresher: rvalue refs

What is an rvalue?

```
int foo = GetInt();

GetInt() = foo;
```

Refresher: rvalue refs

What is an rvalue?

```
int GetInt();

int foo = GetInt();

GetInt() = foo;
```

Refresher: rvalue refs

What is an rvalue?

```
int& GetInt();

int foo = GetInt();

GetInt() = foo;
```

Refresher: rvalue refs

What is an rvalue ref, informally?

Refresher: rvalue refs

What is an rvalue ref, informally?

(Usually) A value without a name, that you couldn’t print in a debugger.

Refresher: rvalue refs

```
void f() {
  GetStrings();  // <- ???
}
```

Refresher: rvalue refs

```
void f() {
  AcceptStrings(GetStrings());
}
```

Refresher: rvalue refs

```
void f() {
  std::vector<std::string> strings = GetStrings();
}
```

Refresher: rvalue refs

```
void f() {
  std::vector<std::string> strings = GetStrings();
  auto more_strings = strings;
}
```

Refresher: rvalue refs

What is an rvalue ref, informally?

(Sometimes) An lvalue that was `std::move`’ed.

Refresher: rvalue refs

What is `std::move`?

A cast to rvalue-reference.

Refresher: rvalue refs

What is `std::move`?

“A name eraser”

Refresher: rvalue refs

```
void f() {
  std::vector<std::string> strings = GetStrings();
  auto more_strings = std::move(strings);
}
```

Refresher: rvalue refs

```
void ZeroNamesIsATemporary() {
  AcceptsStrings(GetStrings());
}

void OneNameIsAMove() {
  std::vector<std::string> strings = GetStrings();
}

void TwoNamesIsACopy() {
  std::vector<std::string> strings = GetStrings();
  auto copy = strings;
}

void AndStdMoveMakesANameNotCount() {
  std::vector<std::string> strings = GetStrings();
  auto not_a_copy = std::move(strings);
}
```

Refresher: rvalue refs

What’s a move c’tor/move assignment op?

Refresher: rvalue refs

What’s a move c’tor/move assignment op?

How a type implements move semantics.

Refresher: rvalue refs

```
class Foo {
 public:
   Foo(const Foo&);      // copy c'tor
   Foo(Foo&&) noexcept;  // move c'tor

   Foo& operator= (const Foo&);      // copy
   Foo& operator= (Foo&&) noexcept;  // move
};
```

Refresher: rvalue refs

What’s a move c’tor/move assignment op?

Move is a source-mutating copy

Refresher: rvalue refs

```
Foo::Foo(Foo&& other)
  : member_(std::move(other.member_)) noexcept {}

Foo& Foo::operator= (Foo&& other) noexcept {
  member_ = std::move(other.member_);
  return *this;
}
```

Refresher: rvalue refs

What’s a forwarding reference?

Refresher: rvalue refs

What’s a forwarding reference?

How you express in templates “take whatever category this was and keep it the same.”

Refresher: rvalue refs

```
template <typename T, typename... Args>
typename memory_internal::MakeUniqueResult<T>::scalar make_unique(
    Args&&... args) {
  return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
}
```

Refresher: rvalue refs

What’s reference qualification?

Refresher: rvalue refs

What’s reference qualification?

Like const-qualification on a method: restrict calls to a method based on the reference-category of the object.

Refresher: rvalue refs

```
class Foo {
 public:
   void Print() & { cout << "lvalue" << endl; }
   void Print() && { cout << "rvalue" << endl; }
};

void f() {
  Foo f;
  f.Print();
  std::move(f).Print();
}
```

A Talk in Three Parts

A Refresher on Rvalue-References

How to use Rvalue-References in API Design

How to use those APIs in Type Design

Good Uses for Rvalue-Refs

Optimization: const-ref + rvalue-ref overload set

```
void std::vector<T>::push_back(const T&);
void std::vector<T>::push_back(T&&);
```

These are everywhere in the standard

Good Uses for Rvalue-Refs

Optimization: Ref qualified member function overload set

```
T& std::optional<T>::value() &;
const T& std::optional<T>::value() const &;
T&& std::optional<T>::value() &&;
const T&& std::optional<T>::value() const &&;
```

Translation: no matter the const-ness or reference category of the `optional`, give me the same version of the underlying `T`.

Good Uses for Rvalue-Refs

Ref qualified member function - Rvalue ref qualified means “steal”

```
std::string std::stringbuf::str() const;
std::string std::stringbuf::str() &&;

std::stringbuf buf;
buf << "Hello World!";
return buf.str();
```

Good Uses for Rvalue-Refs

Ref qualified member function - Rvalue ref qualified means “steal”

```
std::string std::stringbuf::str() const;
std::string std::stringbuf::str() &&;

std::stringbuf buf;
buf << "Hello World!";
return std::move(buf).str();
```

Good Uses for Rvalue-Refs

Or rvalue ref qualified means ”do once”.

Consider a call-once, move-only Callable:

```
std::mfunction<int(std::string)> GetCallable();
void f() {
  GetCallable()("Hello World!");
}
```

Good Uses for Rvalue-Refs

Or rvalue ref qualified means ”do once”.

Consider a call-once, move-only Callable:

```
void f(std::mfunction<int(std::string)> c) {
  std::move(c)("Hello World!");
}
```

Good Uses for Rvalue-Refs

As a parameter, when not an overload set: “maybe move”.

The proposed RCU type (wg21.link/P0561) has

```
bool try_update(const snapshot_ptr<T>& expected,
                std::unique_ptr<T>&& desired);
```

Bad Uses for Rvalue-Refs

As a parameter, when not an overload set: “disallow copies”

```
void Expensive(std::string&& big);
```

Bad Uses for Rvalue-Refs

As a parameter, when not an overload set: “disallow copies”

```
void Expensive(std::string&& big);

std::string my_data = GetData();
Expensive(std::move(my_data));
```

Bad Uses for Rvalue-Refs

As a parameter, when not an overload set: “disallow copies”

```
void Expensive(std::string&& big);

std::string my_data = GetData();
Expensive(std::move(my_data));
Expensive(std::move(my_data));
```

Bad Uses for Rvalue-Refs

As a parameter, when not an overload set: “because optimization”

```
void Cheap(std::string s);
```

or

```
void Cheap(const std::string& s);
void Cheap(std::string&& s);
```

Bad Uses for Rvalue-Refs

As a parameter, in a deleted member of an overload set, to “prevent passing temporaries.”

```
Foo(const std::string& s);
Foo(std::string&& s) = delete;

Foo f("Hello");
```

Bad Uses for Rvalue-Refs

As a parameter, in a deleted member of an overload set, to “prevent passing temporaries.”

```
Foo(const std::string& s);
Foo(std::string&& s) = delete;

{
  std::string hello = "Hello";
  Foo f(hello);
}
```

Bad Uses for Rvalue-Refs

As a parameter, in a deleted member of an overload set, to “prevent passing temporaries.”

```
Foo(const std::string& s);
Foo(std::string&& s) = delete;

{
  std::string hello = "Hello";
  auto f = make_unique<Foo>(hello);
}
```

C++ 11 and on: New Type Designs

Other move-semantics designs:

- move-only types/unique ownership: `std::unique_ptr`

Types that are less-Regular (`std::string_view`)

A Talk in Three Parts

A Refresher on Rvalue-References

How to use Rvalue-References in API Design

How to use those APIs in Type Design

Properties of types

- Invariants
- Thread safety
- Comparable
- Ordered
- Copyable
- Mutable
- Movable

Properties of types - Invariants

Type design is really "What invariants are there on the data members of a T?"

`std::vector` has invariants like:

- capacity >= size
- `data[i]` is a valid T for all i in `[0, size)`
- `data` is a valid / non-null pointer with an allocation of `capacity`

Properties of types - Invariants

Invariants also involve the state model for your type (if any).

Avoid adding states if possible.

- Prefer factory functions or c’tors that throw, rather than T::Init() methods.
- Avoid distinct moved-from states.

Properties of types - Thread Safety

Which operations are safe to call upon a `T` concurrently?

- thread-safe:
  - Concurrent const and non-const operations are OK
- thread-compatible:
  - Concurrent const operations are OK.
  - Any non-const operation requires **all** operations to synchronize
- thread-unsafe:
  - Not even const operations can be invoked concurrently

Properties of types - Comparability

Are operators $==$ and $!=$ defined?

Types - Logical State

There may be a difference between the data members and the logical state of a type.

```
std::string a = "abc";
std::string b;
b.reserve(1000);
b.push_back('a');
b.push_back('b');
b.push_back('c');
assert(a == b);
```

Properties of types - Ordering

Is there a partial or total order for objects of type T?

Which of the operators ==, !=, <, >, <=, and >= are defined?

Properties of types - Ordering

Don’t define Ordering just to put something in a map. If you need a sort order for storage, that’s a property of the storage, not the type.

Ordering depends on the **logical state** of the type.

Properties of types - Copyable

Given a T, can you duplicate its logical state into a new T?

There are two important constraints for copyable types:

- If it is copy-assignable (`operator=`) it should be copy-constructible (a copy constructor). In most cases the reverse is also true.
- The logical state is what is copied.

```
T a = b;
assert(a == b);
```

Properties of types - Mutable

Given a T, can you modify its logical state? In particular, can you modify its state via `operator=`?

Properties of types - Movable

Given a T, can you move its logical state into a new T?

Properties of types - Movable

~~Given a T, can you move its logical state into a new T?~~

`std::is_move_constructible` is equivalent to the following being well-formed:

```
T Foo();


T a = Foo();
```

Regular Types

AKA “value” types - “do what ints do”

- Thread-compatible
- Comparable and ordered
- Copyable, assignable, movable
- Moved-from state?

Example: `std::string`

Structs

Types with no data invariants

Example: `std::pair`

Non-Copyable / Business logic types

These are usually blocks of business logic that hold accessors / handles / streams and perform some business-logic permutations.

- Non-copyable
- Usually non-movable
- Incomparable / unordered

Immutable Types

In situations where an object is shared across many threads concurrently, it may be preferable for all objects of that type to be immutable (after construction).

- Potentially copyable
- Immutable
- Not movable

Reference types

Non-owning, lightweight types that may become invalid because of external changes.

Good for parameters, lightweight representations.

Tricky semantics: careful review of type design strongly suggested.

Example: `std::string_view`, `gsl::span`/`absl::Span`

Move-only types

If your type needs to uniquely represent some resource, move-only semantics may be a good model.

- non-copyable
- Data invariants are guaranteed

Example: `std::unique_ptr`

What’s Next?

- Google Style Guide
- Abseil Tip of the Week
- Updated Core Guidelines?

A Talk in Three Parts

Questions?