Modern C++ API Design, pt 1

Titus Winters (titus@google.com)

Google

Prelude #1

Why do we talk about design?

Prelude #1

Why do we talk about design?

Who does design serve?

Prelude #1

Why do we talk about design?

Who does design serve?

Are we prescriptivist or descriptivist?

Overview, both parts

Micro-API Design

Parameter passing, method qualification, and the importance of overload sets.

Type Properties

What properties can we use to describe types?

Type Families

What combinations of type properties make useful / good type designs?

Overview, both parts

Micro-API Design

Parameter passing, method qualification, and the importance of overload sets.

Type Properties

What properties can we use to describe types?

Type Families

What combinations of type properties make useful / good type designs?

What is the “atom” of C++ API design?

Google

Overload Sets

What does this mean?

```
void f(Foo&& s);
```

Overload sets

A collection of functions in the same scope (namespace, class, etc) of the same name such that if any is found by *name resolution* they all will be.

Overload sets

Per C++ Core Guidelines:

C.162: *Overload operations that are roughly equivalent*

C.163: *Overload only for operations that are roughly equivalent*

Overload sets

Per Google C++ Style:

Use overloaded functions (including constructors) only if a reader looking at a call site can get a good idea of what is happening without having to first figure out exactly which overload is being called.

Good overload sets

Properties of a good overload set:

- Correctness can be judged at the call site without knowing which overload is picked
- A single comment can describe the full set
- Each element of the set is doing “the same thing”

Good overload sets - Varied arity

```
std::string s1 = absl::StrCat("Hello ", name);
std::string s2 = absl::StrCat("Hello ", name, " ", n, " times");
```

Google

Good overload sets - Varied (related) type

Legacy string-ish overloads.

```
void Foo(const char* s);
void Foo(const std::string& s) { Foo(s.c_str()); }
```

Good overload sets - optimization

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

Good overload sets - optimization

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

v.push_back("hello"s);
v.push_back(std::move(world));
```

Good overload sets - optimization

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

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

```
v.push_back("hello"s);
v.push_back(std::move(world));
```

Good overload sets

Properties of a good overload set:

- Correctness can be judged at the call site without knowing which overload is picked
- ***A single good comment can describe the full set***
- Each element of the set is doing “the same thing”

Good overload sets

```
// If calling the version that takes int, returns n + 5.
// Otherwise, prints a friendly message.
void BadOverload(int n);
void BadOverload(const std::string& s);
```

Good overload sets

Properties of a good overload set:

- Correctness can be judged at the call site without knowing which overload is picked
- ***A single good comment can describe the full set***
- Each element of the set is doing “the same thing”

Good overload sets

Most important overload set: copy + move

Good overload sets

When viewing your constructors as an overload set, we start to have a better idea of when “explicit” applies.

Good overload sets

When viewing your constructors as an overload set, we start to have a better idea of when “explicit” applies.

Does a reader of that call site need to know that a (non-copy) constructor was called?

Good overload sets

```
void Fizz(const Foo& f);
void Fizz(const Bar& b);

class Foo {
 public:
  Foo(const Foo& f);
  Foo(const Bar& b);
};
```

Good overload sets

```
void FizzFoo(const Foo& f);
void FizzBar(const Bar& b);

class Foo {
 public:
  Foo(const Foo& f);
  explicit Foo(const Bar& b);
};
```

Bad overload sets

Don’t use =delete in an overload set to describe lifetime requirements.

```
Foo(const std::string& s);
Foo(std::string&& s) = delete;  // No temporaries!

Foo f("Hello");
```

Bad overload sets

Don't use =delete in an overload set to describe lifetime requirements.

```
Foo(const std::string& s);  // s must live until the next call
Foo(std::string&& s) = delete;

Foo f("Hello");
```

Bad overload sets

Don’t use =delete in an overload set to describe lifetime requirements.

Two common cases

- C’tors / setters storing a reference
- Starting async work

Bad overload sets

Don't use =delete in an overload set to describe lifetime requirements.

```
Foo(const std::string& s);
Foo(std::string&& s) = delete;  // No temporaries!

std::string s = "Hello";
Foo f(s);
```

Bad overload sets

Don't use =delete in an overload set to describe lifetime requirements.

```
Foo(const std::string& s);
Foo(std::string&& s) = delete;  // No temporaries!

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

Google

Bad overload sets

```
future<bool> DNAScan(Config, const std::string&) = delete;
future<bool> DNAScan(Config, std::string&&);

auto scan = DNAScan(GetConfig(), GetDNA());
```

Bad overload sets

```
auto scan1 = DNAScan(GetConfig(1), Modify(GetDNA()));
auto scan2 = DNAScan(GetConfig(2), Modify(GetDNA()));
```

Bad overload sets

```
Config c1 = GetConfig(1);
Config c2 = GetConfig(2);
std::string s = Modify(GetDNA());
std::string s2 = s;
// Kick off scans for both configs.
auto scan1 = DNAScan(c1, std::move(s));
auto scan2 = DNAScan(c2, std::move(s2));
```

Bad overload sets

```
future<bool> DNAScan(Config, std::string);
```

```
Config c1 = GetConfig(1);
Config c2 = GetConfig(2);
std::string s = Modify(GetDNA());
// Kick off scans for both configs.
auto scan1 = DNAScan(c1, s);
auto scan2 = DNAScan(c2, std::move(s));
```

Sinks: To Overload or Not To Overload

Is `vector<T>::push_back();` a well-designed overload set? Should everyone accept "sink" parameters in that form?

Sinks: To Overload or Not To Overload

Questions that might affect “How do I express a sink parameter?”

- Is this a generic, or am I sinking a specific type?
- For the type(s) being sunk, how expensive is the function compared to a copy or move for that type?

Sinks: To Overload or Not To Overload

- Generic or specific T?
- Cost of move/copy?
- Are there multiple parameters being sunk?
- Do I know that this will **always** be a sink of exactly T?
- Could allocation reuse dominate?

Sinks: To Overload Or Not To Overload

`const T& + T&&`

If the implementation is small compared to move-constructing T, probably a good design.

More complex. Worse error messages. Worse compilation performance. Combinitorial?

`T`

If the implementation is likely larger cost than move-constructing T, probably good.

Must be sure that this will always be a sink for (exactly) T.

`const T&`

If the implementation is likely larger cost than copy-constructing T, probably good.

Simple, understood, flexible.

Non-Sink Overloads

- `const char* + const string&`

Google™

Non-Sink Overloads

- const char* + const string&
- string_view

Non-Sink Overloads

- const char* + const string&
- string_view
- span

Non-Sink Overloads

```
void f(const std::unique_ptr<Foo>& upf);
void f(const Foo* foo);
void f(const Foo& foo);
```

Non-Sink Overloads

```
void f(std::vector<T*> v);

void f(std::vector<std::unique_ptr<T>> v);

void f(std::vector<std::reference_wrapper<T>> v);
```

Non-Sink Overloads

- const char* + const string&
- string_view
- span
- AnySpan

Non-Sink Overloads

- const char* + const string&
- string_view
- span
- AnySpan
- Concepts?

Non-Sink Overloads

```
void LibraryCall(std::function<void()> f) { f(); }


template <typename Callable>
void LanguageCall(Callable &&f)
{
  std::forward<Callable>(f)();
}
```

Non-Sink Overloads

`std::function` is a simple case

- O(1) thing

Non-Sink Overloads

`std::function` is a simple case

- O(1) thing
- Owning

Non-Sink Overloads vs. Reference Parameters

Only as parameters:

Non-owning reference parameter types are OK, so long as they are only used as function parameters.

Use with caution:

Use non-owning parameter types carefully.

Always question any storage of such a type.

Non-Sink Overloads vs. Reference Parameters

```
// Given "path/to/filename.extension", returns
// the extension (everything after the first '.'
// after the last '/').
// If no '.' is found, or the '.' is the last
// character, returns an empty string_view.
// The returned string_view points into the input.
string_view Suffix(string_view filepath);
```

Non-Sink Overloads vs. Reference Parameters

```
// Given "path/to/filename.extension", returns
// the path (everything before the last '/').
// If no '/' is found, or the '/' is the first
// character, returns an empty string_view.
// The returned string_view points into the input.
string_view Directory(string_view filepath);
```

Non-Sink Overloads vs. Reference Parameters

```
string_view suffix1 = Suffix(path);
string_view suffix2 = Suffix(absl::StrCat("C:\\", path));
auto        suffix3 = Suffix(absl::StrCat("C:\\", path));
string      suffix4 = Suffix(absl::StrCat("C:\\", path));
```

Non-Sink Overloads vs. Reference Parameters

```
string_view suffix1 = Suffix(path);                        // good
string_view suffix2 = Suffix(absl::StrCat("C:\\", path));  // bad
auto        suffix3 = Suffix(absl::StrCat("C:\\", path));  // bad
string      suffix4 = Suffix(absl::StrCat("C:\\", path));  // good
```

Google

Non-Sink Overloads vs. Reference Parameters

Only as parameters:

Non-owning reference parameter types are OK, so long as they are only used as function parameters.

Use with caution:

Use non-owning parameter types carefully.

Always question any storage of such a type.

Overload Sets

Overloads: Not Just Parameters

Critical: Overloading on method qualifiers

Method Qualifier Overloads

Const example

```
      T& std::vector<T>::operator[](size_t ind);
const T& std::vector<T>::operator[](size_t ind) const;
```

Method Qualifier Overloads

Reference example: C++20 stringbuf::str()

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

Rvalue-ref qualifiers can mean “steal”

```
return std::move(buf).str();
```

Google

Method Qualifier Overloads

Const / Ref example:

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

Method Qualifiers

Rvalue-ref qualifiers as “do once”

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

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

Method Qualifiers - Lvalue ref qualifiers

```
struct S {
  S& operator= (const S& rhs) & { return *this; };
};

S ReturnS() { return {}; }
void f() {
  ReturnS() = {};  // ERROR!
}
```

Method Qualifiers - Const

```
class Rotten {
 public:
   void Increment() const { val_++; }
   int val() const { return val_; }

 private:
  mutable int val_ = 0;
};
```

Const vs. Thread Compatibility

Quoth the Standard [res.on.data.races]:

A C++ standard library function shall not directly or indirectly access objects ([intro.multithread]) accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function's arguments, including *this*.

A C++ standard library function shall not directly or indirectly modify objects ([intro.multithread]) accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function's non-const arguments, including *this*.

Google

Const vs. Thread Compatibility

Quoth the Standard [res.on.data.races]:

A C++ standard library function shall not directly or indirectly **access** objects ([intro.multithread]) accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function's arguments, including *this*.

A C++ standard library function shall not directly or indirectly **modify** objects ([intro.multithread]) accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function's **non-const** arguments, including *this*.

Const vs. Thread Compatibility

Quoth the meaning of the standard:

Const accesses to standard types do not cause data races.

Standard types are thread-compatible unless otherwise specified.

Thread Compatible vs. Thread Safe

Thread Compatible:

Concurrent invocation of const methods on this type do not cause data races. Any mutations will require (external) synchronization.

Thread Safe:

Concurrent invocation of methods (const or non-const) on this type do not cause data races.

Const vs. Thread Compatibility

```
class Foo {
 public:
  Response CalculateResponse() const;
 private:
  mutable Response cached_;
};
```

Const vs. Thread Compatibility

```
class Foo {
 public:
  Response CalculateResponse() const;
 private:
  mutable Mutex lock_;
  mutable Response cached_ GUARDED_BY(lock_);
};
```

Questions?
Comments?

Summary Points

- Think in overload sets.
- Move and copy are an overload set
- Make explicit any c’tors that aren’t a “good” overload
- Be skeptical about =delete (and equivalent) in overloads
- string_view etc may replace some overload sets
- Overloading on method qualifiers is powerful
- mutable members are a thread-safety smell
- const methods have thread-safety meaning