Modern C++ API Design, pt 2

Titus Winters (titus@google.com)

Google

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?

Google

Prelude #1

Once upon a time, Euclid came up with the rules for geometry.

Base Rules for Geometry

- Line from 2 points
- Infinite line from finite
- Circle from point+rad
- Right angles are equal

Base Rules for Geometry

- Line from 2 points
- Infinite line from finite
- Circle from point+rad
- Right angles are equal
- Two lines that are parallel to the same line are also parallel to each other.

Base Rules for Geometry

- Line from 2 points
- Infinite line from finite
- Circle from point+rad
- Right angles are equal

- Two lines that are parallel to the same line are also parallel to each other.
    - Or diverge.
    - Or converge

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.

Types

In what ways can the design of a type vary?

Type Properties - Thread Safety

- thread-safe (Good)
- thread-compatible (Good/Default)
- thread-unsafe (Mostly bad)

Type Properties - Comparability ==/!=

- Are == and != defined?

Type Properties - Comparability ==/!=

- Are == and != defined? Only define if you mean “this type is comparable”.

Type Properties - Comparability ==/!=

- Are == and != defined? Only define if you mean "this type is comparable".
- What is the logical state of the type?

Logical State

```
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);
```

Type Properties - Comparability ==/!=

- comparable (Good/default)
- incomparable (Good)

Type Properties - Order <,>,<=,>=

- Is there a (partial or total) order for objects of type T?
- 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.
- Only define <,>,<=,>= if you’re defining an ordering.

Type Properties - Order `<,>,<=,>=`

- Totally ordered (Good)
- Partially ordered - Don't do this as `operator<`
- Unordered (Good)

Type Properties - Copyable

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

There are two important constraints for copyable types:

- operator= implies copy c’tor.
- The logical state is what is copied.

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

Type Properties - Copyable

- Copyable (Good)
- Non-copyable (Good)

Type Properties - Mutable

Given a T, can you modify its logical state?

In particular, can you modify its state via operator=?

Type Properties - Mutable

- Mutable/assignable (Good)
- Immutable (Rare)

Type Properties - Moveable?

Type Properties - Copyable

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

There are two important constraints for copyable types:

- operator= implies copy c'tor.
- The logical state is what is copied.

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

s/copyable/copyable-or-movable/

Google

Type Properties - 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)` - and isn't for any other `data[i]`
- `data` is a valid / non-null pointer with an allocation of `capacity`

Type Properties - Invariants

Invariants are why we have data access restrictions.

Type Properties - Invariants

Core Guidelines: "... if we want to enforce a relation among members, we need to make them private and enforce that relation (invariant) through constructors and member functions".

Google C++ Style Guide: "Make data members private"

Type Properties - Dependent Preconditions

?!?

Type Properties - Dependent Preconditions

Preconditions on vector APIs:

- operator[](size_t ind) - requires ind < size()

Preconditions on optional:

- value() - requires has_value()

Type Properties - Dependent Preconditions

Preconditions on vector APIs:

- operator[](size_t ind) - requires ind < size()

Preconditions on optional:

- value() - requires has_value()

Preconditions on int*:

- operator* - requires that the underlying int is still valid

Type Properties - Dependent Preconditions

Example dependent preconditions:

- int::operator* - requires that the underlying int is still valid
- unique_ptr<T>::reset(T* p) - p isn’t owned by anything else
- “Must hold the lock before calling”
- Must not be called on the UI thread

Type Properties - Dependent Preconditions

Preconditions on vector APIs:

- operator[](size_t ind) - requires ind < size()

Preconditions on optional:

- value() - requires has_value()

Preconditions on int*:

- operator* - requires that the underlying int is still valid
    - Data races?

Google

Type Properties - Dependent Preconditions

Preconditions on **everything**:

- No data races?

```
// This vector isn't shared.
void DoSomething(std::vector<int>& v) {
    ...
}
```

Type Properties - Dependent Preconditions

Preconditions on **everything** - No data races

- No dependent preconditions / remote data (vector, string, int):
  - “This isn’t shared”
  - Default assumption when discussing types
- Dependent preconditions / remote data (int*, string_view):
  - “This + possibly remote data aren’t shared.”

Type Properties - Dependent Preconditions

- int * - does the underlying object still exist?
    - operator==
    - operator*
- string_view - does the underlying buffer still exist?
    - operator[]
    - operator==
- unique_ptr - no other unique_ptr has this non-null value
    - unique_ptr(T*) / reset(T*)

Type Properties - Dependent Preconditions

- No dependent preconditions (Good)
- Dependent preconditions (Warning)

Type Properties

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

Good Type Designs - Regular

Per P0898 -

- Copyable / Movable
- Swappable
- Default constructible
- Assignable
- Comparable

Good Type Designs - Regular

```
const T a = SomeT();
const T b = SomeT();
if (a == b) {
  DoStuff(a);
  assert(a == b);
}
```

Good Type Designs - Regular

```
const int a = SomeT();
const int b = SomeT();
if (a == b) {
  DoStuff(a);
  assert(a == b);
}
```

Google

Good Type Designs - Regular

```
int* const a = SomeT();
int* const b = SomeT();
if (a == b) {
  DoStuff(a);
  assert(a == b);
}
```

Good Type Designs - Regular

```
void DoStuff(int const* r) {
  std::cout << *r << std::endl;
}
int* const a = SomeT();
int* const b = SomeT();
if (a == b) {
  DoStuff(a);
  assert(a == b);
}
```

Good Type Designs - Regular

```
void DoStuff(int const* r) {
  delete r;
}

int* const a = SomeT();
int* const b = SomeT();
if (a == b) {
  DoStuff(a);
  assert(a == b);
}
```

Good Type Designs - Regular

```
void DoStuff(const Rotten& r) {
  r.Increment();
}
const Rotten a = SomeRotten();
const Rotten b = SomeRotten();
if (a == b) {
  DoStuff(a);
  assert(a == b);
}
```

Good Type Designs - Regular

```
const string_view a = SomeT();
const string_view b = SomeT();
if (a == b) {
  DoStuff(a);
  assert(a == b);
}
```

Good Type Designs - Regular

Per P0898 -

- Copyable/Movable
- Swappable
- Default constructible
- Assignable
- Comparable

Good Type Designs - Regular

Per P0898 -

- Copyable/Movable
- Swappable
- Default constructible
- Assignable
- Comparable

Implied

- race free use

Good Geometric Systems

Per Everyone -

- Line from 2 points
- Infinite line from finite
- Circle from point+rad
- Right angles are equal

Euclid

- Two lines that are parallel to the same line are also parallel to each other.

Good Type Designs - Regular

Per P0898 -

- Copyable/Movable
- Swappable
- Default constructible
- Assignable
- Comparable

Implied

- race free use
  - T is unshared
  - T + dependents are unshared
  - Use is single-threaded

Good Type Designs - Regular

```
const T a = SomeT();
const T b = SomeT();
if (a == b) {
  DoStuff(a);
  assert(a == b);
}
```

Good Type Designs - Regular

Per P0898 -

- Copyable/Movable
- Swappable
- Default constructible
- Assignable
- Comparable

Implied

- race free use
  - T is unshared
  - T + dependents are unshared
  - Use is single-threaded

Good Type Designs - Regular

```
void DoStuff(const span<int>& s) {
   s[0]++;
}
const span<int> a = SomeT();
const span<int> b = SomeT();
if (a == b) {
  DoStuff(a);
  assert(a == b);
}
```

Good Type Designs - Regular

```
void DoStuff(const span<int>& s) {
   span<int> copy = s;
   copy[0]++;
}
const span<int> a = SomeT();
const span<int> b = SomeT();
if (a == b) {
  DoStuff(a);
  assert(a == b);
}
```

Good Type Designs - Regular

C++ type design - pick up to 2

- Shallow copy
- Const propagation/deep const
- Deep equality

Good Type Designs - Regular

```
void DoStuff(const span<int>& s) {
   span<int> copy = s;
   copy[0]++;
}
const span<int> a = SomeT();
const span<int> b = SomeT();
if (a == b) {
  DoStuff(a);
  assert(a == b);
}
```

Google

Good Type Designs - Smart References

Just say “No” until

```
std::is_same(const smart_ref<T>,
             smart_ref<const T>)
```

Good Type Designs - Regular

Per P0898 -

- Copyable/Movable
- Swappable
- Default constructible
- Assignable
- Comparable

Implied

- race free use
  - T is unshared
  - T + dependents are unshared
  - Use is single-threaded

Good Type Designs

It’s been put forward that “Any subset of Regular” might be a valid/good design.

Good Type Designs - Semiregular

Per P0898 -

- Copyable/Movable
- Swappable
- Default constructible
- Assignable
- ~~Comparable~~

Good Type Designs - Immutable (for sharing)

Per P0898 -

- Copyable
- ~~Movable, Swappable~~
- Default constructible
- ~~Assignable~~
- Comparable

Good Type Designs - Move Only

Per P0898 -

- ~~Copyable~~
- Movable, Swappable
- Default constructible
- Assignable
- Comparable

~~Good~~ Common Type Designs

Business logic types - behavior not state

- ~~Copyable~~
- Movable?
- Default constructible?
- ~~Assignable~~
- ~~Comparable~~

Good Type Designs - Structs

A struct is a type that has no invariants that must be upheld …

and none of its data members will ever be part of an invariant

(without extensive refactoring).

Good? Type Designs

Non-Owning Reference (Parameter) types?

- Copyable/Movable
- Swappable
- Default constructible
- Assignable
- Comparable ... with dependent preconditions

Open Questions

- What's the future for reference parameters?
- Are we ok with `string_view`?
- How do we design `span`?
- How does this notion of types with dependent preconditions affect how we think about design theory?

Note

See “Revisiting Regular Types” on abseil.io for more

Questions?
Comments?