When Should You Give Two Things the Same Name?

Arthur O’Dwyer

C++ now

2021
MAY 2-7
Aspen, Colorado, USA

When Should You Give Two Things the Same Name?

I also do C++ training!

`arthur.j.odwyer@gmail.com`

Arthur O’Dwyer

2021-05-03

Outline

- When do we use classical inheritance, and why? [4–9]
- Idiosyncratic philosophical digressions [10–16]. Question break.
- Copious anecdotes from the STL [17–28]. Question break.
- Kind of a major rabbit-hole about constructors [29–37]. Question break.
- Mental templates, macros, and polyfills [38–53]. Question break.
- Bonus mantras and takeaways [54–67]. Questions.

Hey look!
Slide numbers!

The proper role of (OO) inheritance

Suppose our codebase contains this class hierarchy.

What should we expect the `Cat`-related codepaths to look like?

```
struct Animal {
    virtual void feed();
    virtual ~Animal() = default;
};

struct Cat : public Animal {
    // probably some members here
};
```

The proper role of (OO) inheritance

Suppose our codebase contains this class hierarchy.
What should we expect the `Cat`-related codepaths to look like?

```
struct Animal {
    virtual void feed();
    virtual ~Animal() = default;
};

struct Cat : public Animal {
    // probably some members here
};
```

We expect to see:

- Some *polymorphic function* `void foo(Animal*)`
- Someplace where `Cat*` is (implicitly) converted to `Animal*`
- Someplace a `Cat` is treated according to its `Animal` *interface*

The proper role of (OO) inheritance

Suppose our codebase contains this class hierarchy.
What should we expect the `Cat`-related codepaths to look like?

```
struct Animal {
    virtual void feed();
    virtual ~Animal() = default;
};

struct Cat : public Animal {
    // probably some members here
};
```

We expect to see:

- Some *polymorphic function* `void foo(Animal*)`
- ...that calls the polymorphic method `animal->feed()`
- ...and/or polymorphically deletes a *heap-allocated* `Animal`

The proper role of (OO) inheritance

If instead we see

```
Cat acquirePet();
void foo(Cat& current) {
    auto newpet = acquirePet();
    std::swap(current, newpet);
}
```

Then we are puzzled!

It's strange to use classical inheritance and virtual member functions when we intend to write value-semantic (non-polymorphic) code.

The proper role of (OO) inheritance

Okay, *sometimes* we use public inheritance without polymorphism:

- EBO — `template<class Alloc> struct Cat : Alloc {~~~}`
- CRTP — `struct Cat : CanFightWith<Cat> {~~~}`
- Tag dispatch — `struct CatTag : AnyAnimalTag {}`

But I’d like to treat these as special cases against a general rule.

Defining `virtual` methods with no polymorphic callers
would be even more surprising.

Liskov Substitution Principle

The “extreme” form of this intuition is related to Barbara Liskov’s Substitution Principle:

- **If** for each object o_1 of type S **there is** an object o_2 of type T **such that** for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, **then** S is a subtype of T.

Adding *intent* to the picture: “Make class S a child of class T **if** you intend to pass an object o_1 of type S as the argument to some function P defined in terms of T.”

If you don’t intend to do that, then...

...plus Occam's Razor

"Make class S a child of class T **if and only if**
you intend to pass an object o_1 of type S
as the argument to some function P defined in terms of T."

If you don't intend to do that, then there is no reason for that public inheritance relationship to exist...

...*and therefore that relationship should* ***not*** *exist*.

William of Ockham:

> "Entities should not be multiplied without reason."

I might add:

> "Techniques should not be introduced without reason."

...plus Chesterton

> The modern reformer says, “I don’t see the use of this fence; let us clear it away.” The more intelligent type answers, “When you can tell me that you ***do*** see why it is here, ***then*** maybe I will allow you to destroy it.”
>
> —G. K. Chesterton (1929), lightly abridged

Since fences generally have reasons,
tearing down fences should not be done lightly.

“Is my refactoring safe?”

...plus Frost

> Before I built a wall I’d ask to know
> What I was walling in, or walling out...
>
> —Robert Frost, “Mending Wall” (1914)

Fences are a kind of “tech debt.”
Per Chesterton, tearing them down is hard.
Therefore, erecting a new fence should be done only for a good reason.

The paradox of the useless fence

Per Chesterton, before we can tear down a “useless” fence, we must understand why it was put up.

If it had no “why” in the first place, it will be harder for us to understand.

Thus: A completely nonsensical fence is harder to remove than one whose rationale is obvious.

My experience of industry codebases is that this happens ***a lot***.

Getting back to C++...

As a reader and maintainer, when we see this class hierarchy,

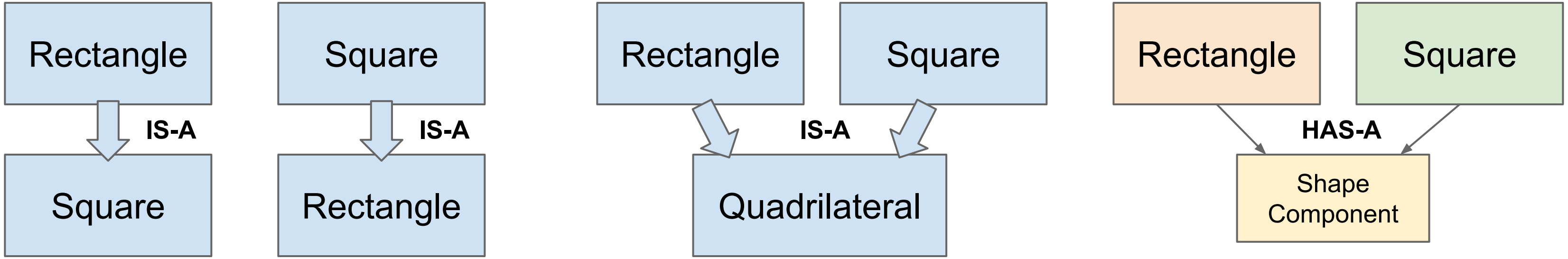
- We expect it to have ***some*** reason (Chesterton’s Fence)
- In fact we expect one ***specific*** reason (Liskov Substitution Principle)

```
struct Animal {
    virtual void feed();
    virtual ~Animal() = default;
};

struct Cat : public Animal {
    // probably some members here
};
```

Many possible approaches

If you ***aren't*** "forced" into one specific inheritance hierarchy by the LSP, then you can imagine many different possible hierarchies...



Rectangle
IS-A
Square

Square
IS-A
Rectangle

Rectangle
Square
IS-A
Quadrilateral

Rectangle
Square
HAS-A
Shape Component

Common wisdom is: Don't introduce an inheritance hierarchy without a reason.
Prefer composition over inheritance.

Questions so far?

A single name for a single entity

In general,

- We should use different words to refer to different ideas.
- When referring to the ***same*** idea, we should use the ***same*** word.

And in C++ specifically,

- Any given single ***identifier*** should refer unambiguously to a single ***entity***.

What do I mean by that...?

A single name for a single entity

Which of these codebases would you prefer to deal with?

A

```
bool feed(Snake& snake);

bool feed(Bear& bear);
```

Or:

B

```
bool feed_snake(Snake& snake);

bool feed_bear(Bear& bear);
```

Both A and B reuse the English ***word*** “feed” —
hopefully to refer to the same ***idea***.

But A alone reuses the same ***identifier*** for two different ***entities***.

A single name for a single entity

Using a distinct identifier for each distinct entity makes a lot of things easier.

- Jumping from call to function definition
    - There’s only one possible destination!
- Grepping for all uses of the entity
- Detecting when an entity has become unused and obsolete
- Globally renaming the entity

When two entities have the same identifier, all these things become harder.

```
    feed(x);  // does this feed a snake or a bear?
```

A single name for a single entity

Tangential: A few things also get easier for the computer.

- Overload resolution
  - There’s only one possible candidate!
- Jump-to-definition in the IDE
  - It can’t jump to the “wrong” entity, because there’s only one!

∴ Overloading should have a reason

When we see this code:

```
bool feed(Snake& snake);
bool feed(Bear& bear);
```

- ***Normally***, two entities should have two names
- We expect the overloading to have some reason (Chesterton’s Fence)
- In fact, I claim, we should expect one ***specific*** reason!
- What is that reason...?

Overloading enables polymorphism

- Classical inheritance enables runtime polymorphism...
- Overloading enables static polymorphism!

```
template<class Animal>
void solve_puzzle(Animal& a) {
    feed(a);
}
```

- When you give two entities the same name, you’re effectively saying
    - “I am going to use this identifier within some kind of ***template***.”

Real example: `push_back`

```
struct Vector { void push_back(int x); };

struct List   { void push_back(int x); };
```

Many different STL containers reuse the identifier `push_back`, ***so that*** the STL can provide `back_inserter`.

```
template<class T>
struct back_insert_iterator {

    ~~~

    back_insert_iterator& operator=(const T::value_type& x) {
        container->push_back(x);

        ~~~

};
```

Real example: `swap`

- Many types overload their own `swap`
- So the identifier `swap` names a ton of different entities
- **This is isomorphic to our `feed` example!**

```
void swap(Snake& a, Snake& b);
void swap(Bear& a, Bear& b);

swap(x, y);  // does this swap two snakes or two bears?
```

- All else equal, I would ***prefer*** to see `swap_snakes(x, y)` ...

Real example: `swap`

- Many types overload their own `swap`
- So the identifier `swap` names a ton of different entities
- **This is isomorphic to our `feed` example!**

```
void swap(Snake& a, Snake& b);
void swap(Bear& a, Bear& b);

swap(x, y);  // does this swap two snakes or two bears?
```

- All else equal, I would ***prefer*** to see `swap_snakes(x, y)`
- But all else is not equal!
    - `swap` is overloaded ***so that*** we can write a templated `std::sort`

Real counter-example: erase

- Each STL container provides two overloads of `erase`

```
class vector {
    using CI = const_iterator;
    iterator erase(CI first, CI last);
    iterator erase(CI where) {
        return erase(where, where+1);
    }
};
```

- One identifier, two entities. What template is enabled by this?

Real counter-example: `erase`

- Having two kinds of `erase` enables no clever template!
- This fence lacks a reason
- And it’s such an awkward fence:

```
std::vector v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

v.erase(std::remove_if(v.begin(), v.end(), isOdd));

assert(std::none_of(v.begin(), v.end(), isOdd));  // fails
```

- “Frost’s Corollary” says we should never have put it up.

Questions so far?

Unfortunate example: Constructors

- Many STL types have *too many* constructor overloads

```
class string {
    string(size_t n, char);
    string(const char *, size_t n);
    string(const string&, size_t pos);
    template<InputIterator It> string(It, It);
};
```

- This fence is pretty annoying...

```
size_t zero = 0;
auto a = std::string(zero, 0);  // Which overload am I calling?
auto b = std::string(0, zero);
```

Unfortunate example: Constructors

- All else being equal, I prefer to replace *most* constructors with factories:

```
class string {
    static string fromCopiesOf(size_t n, char);
    static string fromPtrAndLength(const char *, size_t n);
    static string fromSuffixStartingAt(const string&, size_t pos);
    template<InputIterator It> static string fromRange(It, It);
};
```

- Fence removed?...

```
size_t zero = 0;
auto a = std::string::fromCopiesOf(zero, 0);
auto b = std::string::fromPtrAndLength(0, zero);
```

Constructors are special

- Problem is, all else is *not* equal. C++ loves constructors.
- Factory functions are great, but don't work with C++'s many perfect-forwarding wrappers:
  - `std::make_shared / make_unique`
  - `vector::emplace_back`
  - `optional::emplace / variant::emplace`

```
auto a1 = std::make_shared<std::string>(zero, 0);  // OK

auto a2 = std::make_shared<std::string>(
    std::string::fromCopiesOf(zero, 0));    // ugly; also an extra move
```

Constructors are special

- Constructor syntax is the most convenient way to create an object that's not on the stack.

```
T t1 = T(some, args);
T *p1 = new T(some, args);
```

- Little-known "`new auto`" lets us heap-allocate a factory function's prvalue result:

```
T t2 = T::fromSomeArgs(some, args);
T *p2 = new auto(T::fromSomeArgs(some, args));
```

- "Ugh! How would I write a generic `make_shared` template if `p2`'s syntax were all I had?"— Well, you wouldn't!...

Overloading enables polymorphism

If we want a statically polymorphic `make_shared` template, that can perform the “create” operation on an unknown type `T`, then we must make sure that the “create” operation **has the same name** for all relevant types `T`.

If we want the caller to be able to control the specific “kind of creation” via `make_shared`’s arguments, we must make sure that all the “create” operations for all relevant types `T` accept those arguments in the same way.

```
template<class T, class... Args>
auto build_shared(Args... args) {
    T *p = new auto(T::createGenerically(args...));
    return std::shared_ptr<T>(p);
}
```

Not particularly better, right?

A little way down the rabbit hole

Our fantasy `build_shared` could instead take the “kind of creation” as another parameter.

```
template<class How, class... Args>
auto build_shared(How how, Args&&... args) {
    auto *p = new auto(how(FWD(args)...));
    return shared_ptr(p);
}
```

```
shared_ptr<string> sp = build_shared(string::fromCopiesOf, 0, 0);
shared_ptr<string> sq = build_shared(string::fromPtrAndLength, 0, 0);
```

In today’s C++, this requires `string::fromSomeArgs` to be concrete (not an overload set), so that `make_shared` can deduce that argument’s type

A little way down the rabbit hole

N3617 (Juschka 2013) / P0834 (Dominiak 2017) proposed a core-language syntax for “lifting” an overload set into a concrete lambda object: prefix brackets.

```
auto sp = build_shared([]string::fromCopiesOf, 42, 'x');
```

would mean roughly the same as

```
auto sp = build_shared([](auto&&... args) {
    return string::fromCopiesOf(FWD(args)...);
}, 42, 'x');
```

Sadly, this proposal never went anywhere

A little way down the rabbit hole

P1170 (Revzin 2018) involves more magic. Like `initializer_list`, it’d be visible on the callee side and invisible on the caller side:

```
template<class How, class... Args>
auto build_shared(std::overload_set<How> how, Args&&... args) {
    auto *p = new auto(how(FWD(args)...));
    return std::shared_ptr(p);
}

auto sp = build_shared(string::fromRange, first, last);
```

would deduce its template argument `How` to be the type

```
decltype([](auto&&... a){ return string::fromRange(FWD(a)...); })
```

Questions so far?

To recap so far...

Sometimes we give two entities (in different classes) the same name with the same signature, because we’re going to template on the class type:

```
template<class Animal>
void solve_puzzle(Animal& a) {
    a.feed();
}
```

Sometimes we give two entities (in the same or different classes) the same name with *different* signatures, because we’re going to template on the argument types:

```
template<class Animal, class... Foods>
void solve_infocom_puzzle(Animal& a, Foods... foods) {
    a.feed(foods...);
}
```

This example is isomorphic to standard `make_shared`.

Questionable example: `insert`

All STL containers provide `c.insert(pos, value)`. Associative containers ignore `pos`.

Overloading enables static polymorphism: in this case, generic `insert_iterator`.

```
std::vector<int> data = {3, 1, 4, 1, 5, 9, 2, 6, 5};

std::vector<int> unsorted;
std::copy(data.begin(), data.end(),
          std::inserter(unsorted, unsorted.begin()));

std::set<int> sorted;
std::copy(data.begin(), data.end(),
          std::inserter(sorted, sorted.begin()));
```

Questionable example: `insert`

Just a couple days ago, I heard someone say: “`set.insert(x)` doesn’t always insert. It should really be named `set.insert_or_get(x)`.”

With the STL’s current set of templates, that position is reasonable!

AFAIK, there was never anything in the STL that ***required...***

- the same identifier for `set.insert(x)` and `set.insert(pos, x)`
- the same identifier for `set.insert(x)` and `set.insert(nh)`
- the same identifier for `set.insert(x)` and `multiset.insert(x)`
- the same identifier for `set.insert(x)` and `set.insert(first, last)`

Questionable example: `insert`

“But they all do the same thing, basically! Shouldn’t they have the same name?”

In codebases where you get to choose, I say firmly ***no***.

Sure, use the same ***English word*** for “insert” in each case.

But prefer different C++ ***identifiers*** for different C++ entities *unless you know* why not to. Our code would be clearer and less error-prone with:

- `set.insertAt(pos, x)` versus `set.insert(x)`
- `set.insert(x)` versus `set.insertNodeHandle(nh)`
- `set.insert(x)` versus `set.insertRange(first, last)`

By reusing identifiers instead, we are trading off clarity against... what? When we reuse an identifier, ***we should like to know*** what we are walling in, or walling out.

What about switching containers?

- One of the STL's great benefits is **uniformity of interface**.

```
- std::deque<int> data = {3,1,4,1,5,9,2,6,5};
+ std::vector<int> data = {3,1,4,1,5,9,2,6,5};

  std::sort(data.begin(), data.end());
  auto [first, last] = std::equal_range(data.begin(), data.end(), 5);
  data.push_back(100);
  data.erase(first, last);
  for (int i : data) std::cout << i << '\n';
```

All sequence containers share these APIs!

- We can easily switch our code from `vector` to `deque`, or even `list` or `multiset`
  - e.g. for iterator stability guarantees, or for performance

What about switching containers?

OTOH, it's a real downer that containers can have vastly different semantic properties (re: program correctness) while sharing APIs that are *syntactically* identical.

```
- std::~~deque~~<int> data = {3,1,4,1,5,9,2,6,5};
+ std::vector<int> data = {3,1,4,1,5,9,2,6,5};

  std::sort(data.begin(), data.end());
  auto [first, last] =
      std::equal_range(data.begin(), data.end(), 5);
  data.push_back(100);
  data.erase(first, last);
  for (int i : data) std::cout << i << '\n';
```

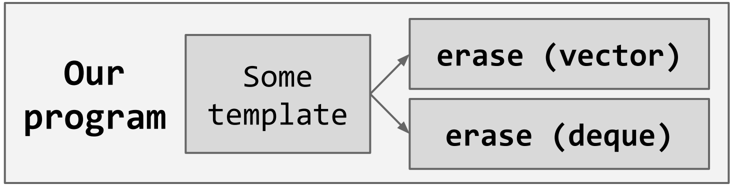
Did you notice that this code's behavior became undefined?

Can templates be mental?

- Titus Winters has said: “Software engineering is programming integrated over time.”

```
std::vector<int> data = {3,1,4,1,5,9,2,6,5};
data.erase(first, last);
```

- The above code demonstrates no reason for `vector.erase(first, last)` and `deque.erase(first, last)` to share the same name. The overloading does not physically enable any static polymorphism.
- No C++ template cares to call `T.erase(first, last)`.

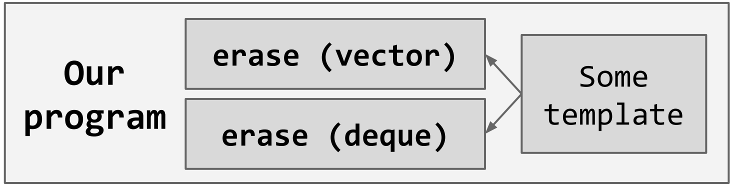


This never happens

Can templates be mental?

- Titus Winters has said: “Software engineering is programming integrated over time.”
- No C++ template cares to call `T.erase(first, last)`...
- But we may “use” the same line of code with `vector` and `deque` at different *times*.

This never happens:



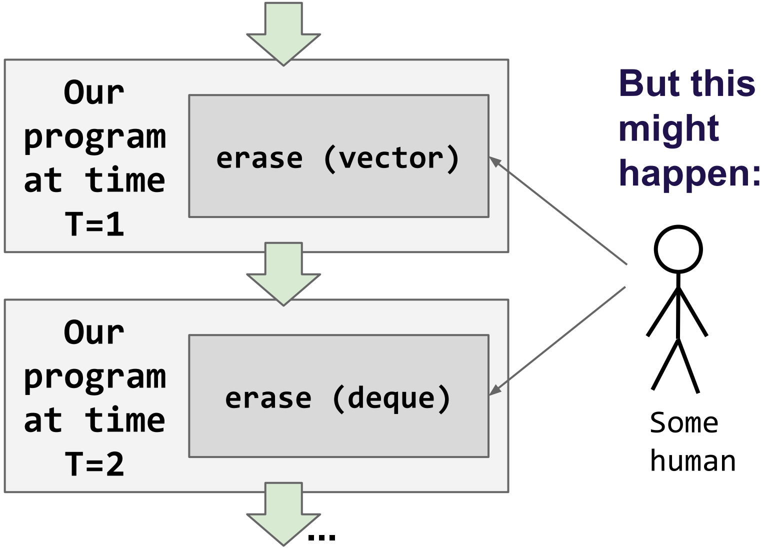
Our program

erase (vector)

erase (deque)

Some template

But this might happen:



Our program at time T=1

erase (vector)

Our program at time T=2

erase (deque)

...

Some human

Sharing names as an upgrade path

- The STL reuses names between `string` and `string_view`.

```
std::string trimmed_name(const std::string& filename) {
  if (filename.ends_with(".png") || filename.ends_with(".jpg")) {
    return filename.substr(0, filename.size() - 4);
  }
  return filename;
}
```

- `string::substr` and `string_view::substr` are entities with the same name
- Ditto `string::ends_with` and `string_view::ends_with`
- Do we have a template that calls `substr` and is instantiated for both `string` and `string_view`? **No.** But we may refactor this code ***from*** `string` ***to*** `string_view`.

Sharing names as an upgrade path

- The “Chesterton’s Fence” reason for giving two entities the same identifier in this case is to let us refactor this code from `string` to `string_view`.

```
- ~~std::string~~ trimmed_name(~~const std::string&~~ filename) {
+ std::string_view trimmed_name(std::string_view filename) {
    if (filename.ends_with(".png") || filename.ends_with(".jpg")) {
      return filename.substr(0, filename.size() - 4);
    }
    return filename;
  }
```

All “stringy” types share these APIs!

Sharing names as an upgrade path

This was arguably a reason `std::optional` got the `*` and `->` operators.

```
  struct Connection {
    void load_dummy_certificate() {
-     cert_ = std::~~make_unique~~<Certificate>("example.com");
+     cert_ = std::make_optional<Certificate>("example.com");
    }
    void connect() {
        if (cert_) setCiphers(cert_->ciphers());
        ~~~
    }
-   std::unique_ptr<Certificate> cert_;
+   std::optional<Certificate> cert_;
  };
```

Of course mainly `std::optional` got them from `boost::optional`. But then why did `boost::optional` have them?... Upgrade paths!

Reusing names still leads to bugs

However, in another context, `optional` can be a poster child for how it’s *still, by default, a bad idea* to reuse a name without good reason.

```
struct DataCache { void update(Key, Value); void reset(); };

struct Connection {
    std::optional<DataCache> dataCache_;

    void resetCache() {
        if (dataCache_) {
            dataCache_.reset();
        }
    }
};
```

See the bug?

Reusing names still leads to bugs

However, in another context, `optional` can be a poster child for how it's *still, by default, a bad idea* to reuse a name without good reason.

```
struct DataCache { void update(Key, Value); void reset(); };

struct Connection {
    std::optional<DataCache> dataCache_;

    void resetCache() {
        if (dataCache_) {
            dataCache_.reset();
        }
    }
};
```

It turns out that the programmer intended to write

```
dataCache_->reset();
```

to call `DataCache::reset`. But what they actually wrote called `std::optional<DataCache>::reset`, which called `~DataCache()`!

I would lay the blame here on `DataCache::reset()`; it should not have the same name as `optional::reset` (nor `unique_ptr::reset`).

Other variations on “upgrade path”

- The members of `std::optional` are mostly named the same as the members of `std::unique_ptr`, because upgrade paths.
- But really, they’re named the same as the members of ***`boost::optional`***... because upgrade paths!

```
#if __cplusplus >= 201703L
 #include <optional>
 using std::optional;
#else
 #include <boost/optional.hpp>
 using boost::optional;
#endif
```

The APIs of these two types are largely identical, so that the same codebase can work with either one.

It’s not C++-template-based static polymorphism, but it is “macro-based static polymorphism.”

Specifically, here we are providing a ***polyfill*** for C++17 `optional`.

Other variations on “upgrade path”

- Another reason for “macro-based static polymorphism” is platform-specific code.

```
namespace curses {
  void clearScreen();
  void drawAt(int x, int y, char ch);
}

namespace conio {
  void clearScreen();
  void drawAt(int x, int y, char ch);
}
```

It’s important that these two functions `drawAt` have the same name. Changing their names to `drawWithCurses` and `drawWithConio` would break the code in `drawTitleBar()`.

```
using namespace TERMLIB;  // -DTERMLIB=curses or -DTERMLIB=conio

void drawTitleBar() { for (int x=0; x < 100; ++x) drawAt(x, 1, '#'); }
```

Questions so far?

What about default arguments? 👿

In “Back to Basics: Algebraic Data Types” (CppCon 2020),
I gave this example of what **not** to do:

```
void openConnection(std::string_view host,
                    std::optional<Cert> cert = std::nullopt);
```

```
openConnection("example.com", Cert("foo"));   // OK, HTTPS
openConnection("example.com");                // OK, HTTP (implicitly)
openConnection("example.com", std::nullopt);  // OK, HTTP (explicitly)
```

This API looks attractive at first glance.

However, in a real codebase, I’d expect exactly one of two situations...

What about default arguments? 👿

Common Pattern 1: `openConnection` is an “internal implementation” kind of API.
All its callers are passing along an `optional<Cert>` they got from somewhere else.
Nobody ever actually calls it with just a single argument.
The default is completely redundant.

Prefer:

```
void openConnection(std::string_view host,
                    const std::optional<Cert>& cert);
```

```
openConnection("example.com", optCert);   // if optCert is engaged, HTTPS
openConnection("example.com", optCert);   // otherwise, HTTP
```

I see this “totally unused default value” antipattern frequently in Python, too.

What about default arguments? 👿

Common Pattern 2: `openConnection` is a “user-facing” kind of API.
All its callers are “business logic.”
Each knows whether it wants to open an HTTP connection or an HTTPS connection.
The ability to quietly “forget” the cert parameter is a *massive liability*.
This is isomorphic to our example with `std::vector::eraseAt / eraseRange`!

This is two different jobs, which should be two different entities with two different names.

```
    void openInsecureConnection(std::string_view host);

    void openSecureConnection(std::string_view host,
                              const Cert& cert);

        openSecureConnection("example.com", Cert("foo"));  // HTTPS
        openInsecureConnection("example.com");             // HTTP
```

What about default arguments? 👿

Notice that the two common patterns complement each other.

```
void openConnectionImpl(std::string_view host, const Cert *cert);

void openInsecureConnection(std::string_view host) {
    return openConnectionImpl(host, nullptr);
}

void openSecureConnection(std::string_view host, const Cert& cert) {
    return openConnectionImpl(host, &cert);
}
```

A function with defaulted arguments behaves a lot like an overload set —
i.e. it behaves a lot like multiple entities all with the same name —
i.e. it is Frost’s fence: It must be justified.

I have a blog post you should read

“Given general function overloading, default arguments
are logically redundant and at best a minor notational convenience.
However, C with Classes had default argument lists for years
before general overloading became available in C++.”

—Bjarne Stroustrup, *D&E* (1994), section 2.12.2

“Default function arguments are the devil.”

—Arthur O’Dwyer

https://quuxplusone.github.io/blog/2020/04/18/default-function-arguments-are-the-devil/

Another relevant mantra

> Although Concepts are constraints on types,
> you don't find them by looking at the types in your system.
> You find them by studying the algorithms.
>
> — @ericniebler

This is essentially what we've been doing throughout this talk.

When deciding whether to give two entities the same name,
look at ***how they are going to be used***, i.e., study the algorithms.

If the algorithm (caller) doesn't need them to have the same name,
then don't give them the same name.

Bonus: Enumerators?

When we give class members or free functions the same names,
we can exploit that through templates.

When we give enumerators the same names, technically, we can exploit that through templates as well. But I’ve never seen this in real life...

```
enum class NetworkError { Okay, NoSuchHost, NoRouteToHost };
enum class DiskError { Okay, NoSuchFile, WrongPermissions };
void printFileFromURI(auto& getter, std::string_view URI) {
    auto [contents, error] = getter.getContents(URI);
    if (e == decltype(e)::Okay) {
        std::cout << contents << '\n';
    }
}
```

Takeaways

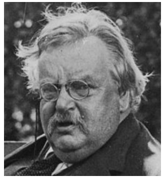
- Inheritance is for sharing an interface (Liskov Substitution Principle)
  - ...and so is overloading!

Takeaways

- Inheritance is for sharing an interface (Liskov Substitution Principle)
    - ...and so is overloading!
- Use a single name for a single entity (Occam's Razor -ish)

Takeaways

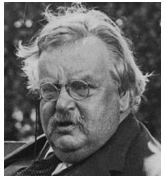
- Inheritance is for sharing an interface (Liskov Substitution Principle)
    - ...and so is overloading!
- Use a single name for a single entity (Occam's Razor -ish)
- When you see two things with the same name,
  assume there is a reason for it (Chesterton's Fence)



Takeaways

- Inheritance is for sharing an interface (Liskov Substitution Principle)
  - ...and so is overloading!

- Use a single name for a single entity (Occam's Razor -ish)

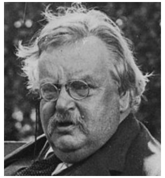


- When you see two things with the same name,
  assume there is a reason for it (Chesterton's Fence)

- When you have the option to give two things the same name,
  don't, unless there is a reason for it (Frost's Corollary)

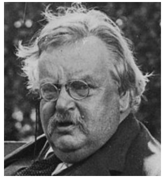
Takeaways

- Inheritance is for sharing an interface (Liskov Substitution Principle)
  - ...and so is overloading!
- Use a single name for a single entity (Occam's Razor -ish)
- When you see two things with the same name, assume there is a reason for it (Chesterton's Fence)
- When you have the option to give two things the same name, *don't, unless* there is a reason for it (Frost's Corollary)
- To find concepts, don't study what your callees provide in common; study what your callers require



Takeaways

- Inheritance is for sharing an interface (Liskov Substitution Principle)
    - ...and so is overloading!
- Use a single name for a single entity (Occam's Razor -ish)
- When you see two things with the same name, assume there is a reason for it (Chesterton's Fence)
- When you have the option to give two things the same name, *don't, unless* there is a reason for it (Frost's Corollary)
- To find concepts, don't study what your callees provide in common; study what your callers require
- Default function arguments are the devil



Questions?