A Type, by Any Other Name

Jon Cohen

Hi there! I’m Jon!

Yay!

Agenda

Renaming Types

Motivation

Examples

Conclusion

Renaming Types

Renaming Types

Before

```
// common.h
class Old {};

void f(Old) {}

// user.h
void g() {
  Old foo;
  f(foo);
}
```

After

```
// common.h
class New {};

void f(New) {}

// user.h
void g() {
  New foo;
  f(foo);
}
```

Motivation

Why Rename a Type?

- To upgrade it
- To move it
- To fix dependency issues
- To allow non-atomic codebase refactoring

Why?!

Why Non-Atomic Refactoring?

- Changes may be too large to merge
- Changes may involve excessive coordination
- Large changes introduce extra complexity
- Changes may affect disparate repos

Renaming Types - New Type

```
class Old {};

class New {};

// Migrate users
```

- Great in many cases
- Requires migrating entire call chains
- Rewrites are expensive / risky

Renaming Types - Conversions

```
class Old {};

class New {
  New(const Old&);
  operator Old();
};
```

- Problematic if Old or New are expensive to copy
- Not exact equivalence

Renaming Types - Conversions

```
class Old {};

class New {
  New(const Old&);
  operator Old();
};

void f(const std::vector<Old>&) {}

void g(const std::vector<New>& v) {
  f(v);
}
```

- Problematic if Old or New are expensive to copy
- Not exact equivalence

Renaming Types - Aliases

```
class New {};

using Old = New;
```

- Aliases are literally the same type.

Renaming Types - Aliases

```
class New {};

using Old = New;

void f(const std::vector<Old>&) {}

void g(const std::vector<New>& v) {
  f(v);
}
```

- Aliases are literally the same type.

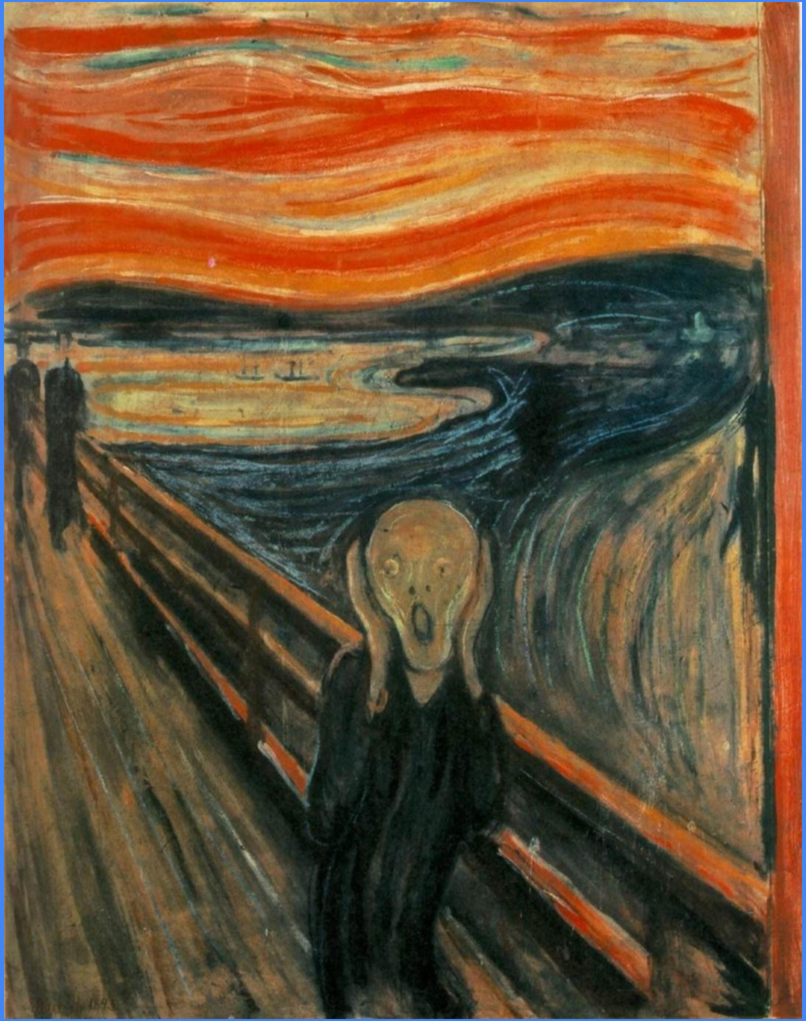
```
T& tref = t     : T tcopy = t ::
using alias = T : conversion
```

Examples

Just Kidding...

Take a Deep Breath…

ADL



ADL - Argument Dependent Lookup

```
namespace n {
class Class {};
void TakesClass(Class) {}
}
```

ADL - Argument Dependent Lookup

```
namespace n {
class Class {};
void TakesClass(Class) {}
}

void f() {
  n::Class c;
  // calls n::TakesClass;
  TakesClass(c);
}
```

ADL - Argument Dependent Lookup

```
namespace n {
class Class {};
void swap(Class&, Class&) {}
}
```

ADL - Argument Dependent Lookup

```
namespace n {
class Class {};
void swap(Class&, Class&) {}
}

template <typename T>
void Swap(T& a, T& b) {
  using std::swap;
  swap(a, b);
}
```

- `Swap<int>` calls `std::swap`
- `Swap<n::Class>` calls `n::swap`

ADL couples types and functions

Examples

Example - Changing Name

Before

```
// common.h
class Old {};

void f(Old) {}

// user.h
void g() {
  Old foo;
  f(foo);
}
```

After

```
// common.h
class New {};
using Old = New;

void f(New) {}

// user.h
void g() {
  Old foo;
  f(foo);
}
```

Example - Changing a Namespace

Before

```
// common.h
namespace old {
class Class {};
void f(Class) {}
}

// user.h
void g() {
  old::Class foo;
  old::f(foo);
}
```

Example - Changing a Namespace

Before

```
// common.h
namespace old {
class Class {};
void f(Class) {}
}

// user.h
void g() {
  old::Class foo;
  old::f(foo);
}
```

After

```
// common.h
namespace absl {
  class Class {}
  void f(Class) {}
}

namespace old {
using absl::Class;
using absl::f;
}
```

Example - Changing a Namespace

Before

```
// common.h
namespace old {
class Class {};
void f(Class) {}
}

// user.h
void g() {
  old::Class foo;
  f(foo);
}
```

After

```
// common.h
namespace absl {
  class Class {}
  void f(Class) {}
}

namespace old {
using absl::Class;
using absl::f;
}
```

Example - Changing a Namespace

Before

```
// common.h
namespace old {
class Class {};
void f(Class) {}
}

// user.h
void g() {
  old::Class foo;
  f(foo);
}
```

After

```
// common.h
namespace absl {
  class Class {}
}

namespace old {
using absl::Class;
void f(Class) {}
}
```

Example - Changing a Namespace

Before

```
// common.h
namespace old {
class Class {};
void f(Class) {}
}

// user.h
void g() {
  old::Class foo;
  f(foo);
}
```

error: no matching function call to ‘f’

After

```
// common.h
namespace absl {
  class Class {}
}

namespace old {
using absl::Class;
void f(Class) {}
}
```

How to Make a Puppy Sad

- Call code you don’t own via ADL
- Use the global namespace

Example - Changing Name and Namespace

Before

```
// common.h
namespace old {
class Old {};
void f(Old) {}
}

// user.h
void g() {
  old::Old foo;
  old::f(foo);
}
```

After

```
// common.h
namespace absl {
  class New {}
  void f(New) {}
}

namespace old {
using Old = absl::New;
using absl::f;
}
```

Example - Changing Name and Namespace

Before

```
// common.h
namespace old {
class Old {};
void f(Old) {}
}

// user.h
namespace old {
class Old;
}
```

After

```
// common.h
namespace absl {
  class New {}
  void f(New) {}
}

namespace old {
using Old = absl::New;
using absl::f;
}
```

Example - Changing Name and Namespace

Before

```
// common.h
namespace old {
class Old {};
void f(Old) {}
}

// user.h
namespace old {
class Old;
}
```

error: definition of type 'Old' conflicts with type alias of the same name

After

```
// common.h
namespace absl {
  class New {}
  void f(New) {}
}

namespace old {
using Old = absl::New;
          bsl::f;
}
```

How to Make a Panda Sad

- Call code you don’t own via ADL
- Use the global namespace

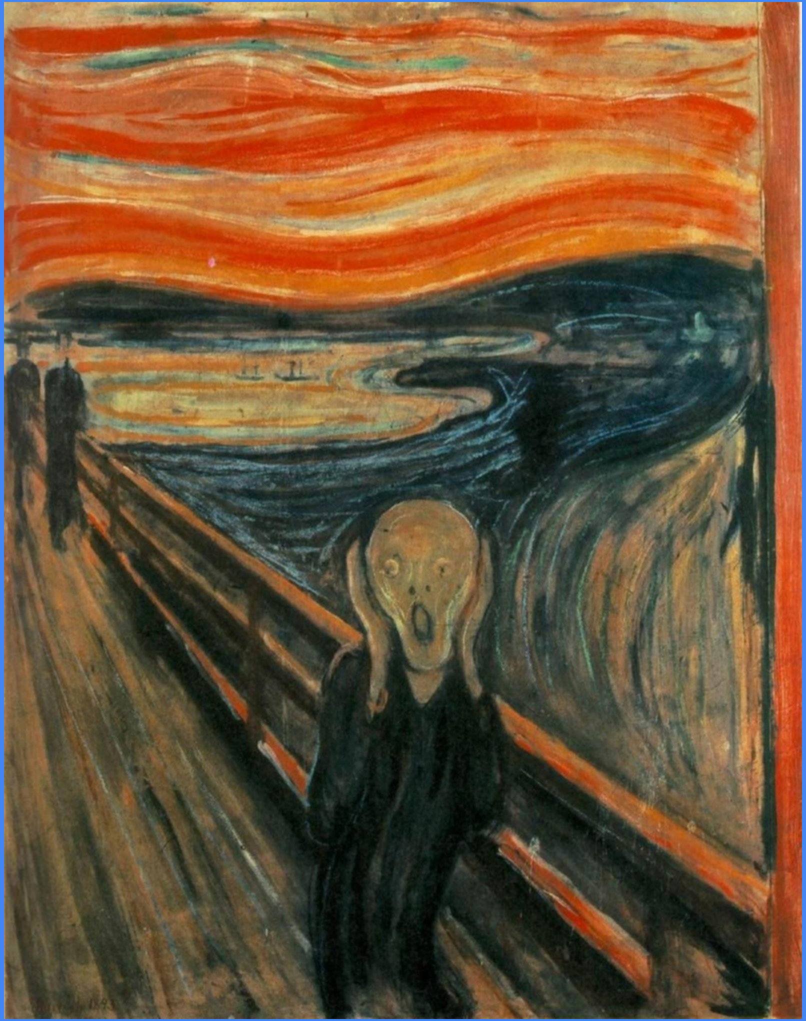
How to Make a Panda Sad

- Call code you don’t own via ADL
- Use the global namespace
- Forward-declare a type you don’t own

How to Make a Panda Sad

- Call code you don’t own via ADL
- Use the global namespace
- Forward-declare a type you don’t own
- Open a namespace you don’t own

What about templates?



Renaming Types - Aliases

```
template <typename T>
class New {};

template <typename T>
using Old = New<T>;
```

Renaming Types - Aliases

```
template <typename T>
class New {};

template <typename T>
using Old = New<T>;

namespace n {
using ::New;
}
```

Renaming Types - Aliases

```
template <typename T>
class New {};

template <typename T>
using Old = New<T>;

namespace n {
using ::New;
}

template <typename T>
using Metafunction = New<const T>;
```

Example - Renaming a Template

Before

```
// common.h
namespace old {
template <typename T>
internal::Ret<T> f();
}

// user.h
struct S {
  template <typename T>
  friend old::internal::Ret<T>
  old::f();
};
```

After

```
// common.h
namespace absl {
template <typename T>
internal::Ret<T> f;
}

namespace old {
using absl::f;
}
```

Example - Renaming a Template

Before

```
// common.h
namespace old {
template <typename T>
internal::Ret<T> f();
}
```

error: no member named 'internal' in namespace 'old';

```
//
st
  template  typename T>
  friend old::internal::Ret<T>
  old::f();
};
```

After

```
// common.h
namespace absl {
template <typename T>
internal::Ret<T> f;
}

namespace old {
using absl::f;
}
```

How to Make an Owl Sad

- Call code you don’t own via ADL
- Use the global namespace
- Forward-declare a type you don’t own
- Open a namespace you don’t own

How to Make an Owl Sad

- Call code you don’t own via ADL
- Use the global namespace
- Forward-declare a type you don’t own
- Open a namespace you don’t own
- Name an internal type you don’t own

How to Make an Owl Sad

- Call code you don’t own via ADL
- Use the global namespace
- Forward-declare a type you don’t own
- Open a namespace you don’t own
- Name an internal type you don’t own
- Specify deducible type parameters

Don’t Rely on Implementation Details of Code You Don’t Own

Dependent Types

Dependent Types

```
template<typename T>
using Dependent = typename std::remove_const<T>::type;
```

Dependent Types

```
template<typename T>
using Dependent = typename std::remove_const<T>::type;

template <typename T>
using AlsoDependent = std::remove_const_t<T>;
```

Template parameters of dependent types can’t be deduced

Merging Two Types

Why Merge Types?

- Many similar hand-rolled types with different interfaces
- Facilitate type migration
- Abstract away semantic differences

Merging Template Interface Types

```
template <typename T>
class ArraySlice {
  // ...
  template <typename C>
  ArraySlice(const C&);
  // ...
  const T& operator[](int);
};
```

```
template <typename T>
class MutableArraySlice {
  // ...
  template <typename C>
  MutableArraySlice(C*);
  // ...
  T& operator[](int);
};
```

Span<T>

Merging Template Interface Types

```
template <typename T>
using Span = std::conditional_t<
  std::is_const_v<T>, ArraySlice<T>, MutableArraySlice<T>>;
```

Merging Template Interface Types

```cpp
template <typename T>
using Span = std::conditional_t<
  std::is_const_v<T>, ArraySlice<T>, MutableArraySlice<T>>;

// user.h
template <typename T>
void TakesSpan(Span<T>) {}

void f (Span<int> s) {
  TakesSpan(s);
}
```

Merging Template Interface Types

```
template <typename T>
using Span = std::conditional_t<
  std::is_const_v<T>, ArraySlice<T>, MutableArraySlice<T>>;

// user.h
template <typename T>
void TakesSpan(Span<T>) {}

void f (Span<int> s) {
  TakesSpan(s);
}
```

note: candidate template ignored: couldn't infer template argument 'T'

error: no matching function call to ‘TakesSpan’

Merging Template Interface Types

```cpp
template <typename T>
class Span {
  using Impl = std::conditional_t<
      std::is_const_v<T>,
      ArraySliceImpl<T>, MutableArraySliceImpl<T>>;
};

using ArraySlice = Span<const T>;
using MutableArraySlice = Span<T>;
```

Aliases are a tool for gradual, non-atomic refactoring

Thank You!