010 Tricks that only Library Implementers Know!

Marshall Clow[1], Jonathan Wakely[2]

[1]Qualcomm

[2]Red Hat

April 14, 2018

About Jonathan and Marshall

Jonathan Wakely is the lead developer for libstdc++, the standard library implementation that ships with GCC.

Marshall Clow is the lead developer for libc++, the standard library implementation that ships with Clang.

About Jonathan and Marshall

Jonathan Wakely is the lead developer for libstdc++, the standard library implementation that ships with GCC.

Marshall Clow is the lead developer for libc++, the standard library implementation that ships with Clang.

What do we mean 'tricks'?

Techniques that we don't see commonly used, but we think are generally useful.

Empty Base class optimizations

When you're writing generic code, you often need to store objects whose types you don't know until later.

Sometimes these objects are small - so small, in fact, that they have no state.

1. `unique_ptr` and `shared_ptr` have a deleter, `std::default_deleter` by default.
2. `set` and `map`, as well as many algorithms, use `std::less` by default.
3. All the containers (except `array`) have an allocator - `std::allocator` by default.

Empty Base class optimizations (2)

If these objects are empty (have no state), there's no reason to store them. You can just construct one whenever you need it - they're all the same.

All of the standard library implementations that I've checked have an internal class (or facility) named (something like) `compressed_pair`, which holds two objects. The difference between this and `std::pair` is that `compressed_pair` doesn't actually store the objects if they are empty.

in C++2a, we will have the attribute `[[no_unique_address]]` as a compiler-based solution.

Why is this important?

If you store them, they take up space in the object - every member has a unique address.

This could double the size of a `unique_ptr`, for example.

Constraining "greedy templates"

- Some types can be constructed from nearly anything:

```
struct any {
  template<typename T>
    any(T&&);
    ...
};
```

Constraining "greedy templates"

- Some types can be constructed from nearly anything:

```
struct any {
  template<typename T>
    any(T&&);
    ...
};
```

- But we don't want it to accept *anything*:

```
any a1;
any a2 = a1;
```

Constraining "greedy templates"

- Some types can be constructed from nearly anything:

```
struct any {
  template<typename T>
    any(T&&);
    ...
};
```

- But we don't want it to accept *anything*:

```
any a1;
any a2 = a1;
```

- Here `a1` is a non-const lvalue, so overload resolution chooses `any(T&&)` over `any(const any&)`.

Constraining “greedy templates”

Put a SFINAE constraint on it!

```
struct any {
  template<typename T,
           typename = enable_if_t</*???*/>>
    any(T&&);
    ...
};
```

Constraining "greedy templates"

Put a SFINAE constraint on it!

```
struct any {
  template<typename T,
           typename = enable_if_t<!is_same_v<decay_t
    any(T&&);
    ...
};
```

Constraining "greedy templates"

Put a SFINAE constraint on it!

```
struct any {
  template<typename T, typename = enable_if_t<
                !is_same_v<decay_t<T>, any>>>
    any(T&&);
    ...
};
```

Constraining "greedy templates"

Slightly better to use C++2a's remove_cvref instead of decay:

```
struct any {
  template<typename T, typename = enable_if_t<
            !is_same_v<remove_cvref_t<T>, any>>>
    any(T&&);
    ...
};
```

Using common_type_t<T> as an identity meta-function

- If you have some template metaprogramming that wants to apply a transformation:

```
template<T, template<typename...> class F>
  using transformed_t = typename F<T>::type;
template<typename T>
  struct second {
    using type = typename T::second_type;
  };
static_assert(std::is_same_v<char,
  transformed_t<std::pair<int, char>, second>>);
```

Using common_type_t<T> as an identity meta-function

- If you have some template metaprogramming that wants to apply a transformation:

```
template<T, template<typename...> class F>
  using transformed_t = typename F<T>::type;
template<typename T>
  struct second {
    using type = typename T::second_type;
  };
static_assert(std::is_same_v<char,
  transformed_t<std::pair<int, char>, second>>);
```

- ...an "identity" meta-function is useful for cases where nothing needs transforming:

```
template<typename T>
  struct identity { using type = T; };
static_assert(std::is_same_v<char,
  transformed_t<char, identity>>);
```

Using common_type_t<T> as an identity meta-function

But it's also useful to create a *non-deduced context*.

```
template<typename T>
  void frob(std::vector<T>& a, T b)
  { for (auto& c : a) c *= b; }

std::vector<long> a{1, 2, 3};
frob(a, 5);
```

```
error: no matching function for call to
'frob(std::vector<long int>&, int)'

note: template argument deduction/substitution failed:
note: deduced conflicting types for parameter 'T' ('long
int' and 'int')
```

Using common_type_t<T> as an identity meta-function

The problem is that the second function parameter participates in argument deduction:

```
template<typename T>
  void frob(std::vector<T>& a, T b);
```

So if you call it with `vector<long>` and `int` the compiler can't deduce `T`, because it deduces `long` from the first argument and `int` from the second.

```
note: deduced conflicting types for parameter 'T' ('long
int' and 'int')
```

Using common_type_t<T> as an identity meta-function

The solution is to ensure the second argument is a *non-deduced context*:

```
template<typename T>
  void frob(std::vector<T>& a,
            typename identity<T>::type b);
```

Here the second use of T is a non-deduced context, so only the first one participates in argument deduction. That allows T to be deduced from vector<long>, and then long is substituted into identity<T>::type, which gives long.
The argument 5 can be converted to long, so the call compiles.

Using common_type_t<T> as an identity meta-function

Instead of defining a class template `identity` just for cases like these you can use `std::common_type` with a single template argument:

```
static_assert(std::is_same_v<char,
  transformed_t<char, std::common_type>>);

template<typename T>
  void frob(std::vector<T>& a,
            std::common_type_t<T> b);

template<typename T>
  using identity = std::common_type<T>;
template<typename T>
  using identity_t = std::common_type_t<T>;
```

Conditionally deleted special members

- When you have a generic wrapper type like `std::optional` you want the wrapper type to model the same interface as the object it contains.

Conditionally deleted special members

- When you have a generic wrapper type like `std::optional` you want the wrapper type to model the same interface as the object it contains.
- If `is_copy_constructible<T>` is `false` then you also want `is_copy_constructible<optional<T>>` to be `false`.
- If `is_default_constructible<T>` is `false` then you also want `is_default_constructible<optional<T>>` to be `false`.
- ...

Conditionally deleted special members

- The usual trick for conditionally deleting functions is SFINAE, but you can't use that here.

```
template<typename T>
  struct optional {
    template<typename U,
             typename = enable_if_t<
               is_same_v<U, optional>
               && is_copy_constructible_v<U>>>
      optional(const U&);
```

Conditionally deleted special members

- The usual trick for conditionally deleting functions is SFINAE, but you can't use that here.

```
template<typename T>
  struct optional {
    template<typename U,
             typename = enable_if_t<
               is_same_v<U, optional>
               && is_copy_constructible_v<U>>>
      optional(const U&);
```

- This template isn't a copy constructor.

Conditionally deleted special members

- The usual trick for conditionally deleting functions is SFINAE, but you can't use that here.

```
template<typename T>
  struct optional {
    template<typename U,
             typename = enable_if_t<
               is_same_v<U, optional>
               && is_copy_constructible_v<U>>>
      optional(const U&);
```

- This template isn't a copy constructor.
- So the compiler will still generate a copy constructor implicitly!

Conditionally deleted special members

- The solution is to define it as defaulted and get the compiler to delete it for us when appropriate:

```
template<typename T>
  struct optional {
    optional(const optional&) = default;
```

Conditionally deleted special members

- The solution is to define it as defaulted and get the compiler to delete it for us when appropriate:

```
template<typename T>
  struct optional {
    optional(const optional&) = default;
```

- But how do we get the compiler to delete it?

Conditionally deleted special members

- The solution is to define it as defaulted and get the compiler to delete it for us when appropriate:

```
template<typename T>
  struct optional {
    optional(const optional&) = default;
```

- But how do we get the compiler to delete it?
- Delegate the decision to a base class:

```
template<typename T>
  struct optional
  : maybe_copyable<is_copy_constructible_v<T>>
  {
    optional(const optional&) = default;
```

Conditionally deleted special members

```
template<bool IsCopyable>
  struct maybe_copyable { };

template<>
  struct maybe_copyable<false> {
    maybe_copyable(const maybe_copyable&) = delete;
  };

template<typename T>
  struct optional
  : maybe_copyable<is_copy_constructible_v<T>>
  {
    optional(const optional&) = default;
    ...
  };
```

Conditionally deleted special members

```
template<bool IsCopyable>
  struct maybe_copyable { };

template<>
  struct maybe_copyable<false> {
    maybe_copyable(const maybe_copyable&) = delete;
    // default the rest so they aren't disabled:
    maybe_copyable() = default;
    maybe_copyable(maybe_copyable&&) = default;
    maybe_copyable&
    operator=(const maybe_copyable&) = default;
    maybe_copyable&
    operator=(maybe_copyable&&) = default;
  };
```

Conditionally deleted special members

Then you simply repeat this for each special member:

```
template<typename T>
  struct optional
  : maybe_copyable<is_copy_constructible_v<T>>,
    maybe_movable<is_move_constructible_v<T>>,
    maybe_copy_assignable<is_copy_assignable_v<T>>,
    maybe_move_assignable<is_move_assignable_v<T>>
  {
    optional(const optional&) = default;
    optional(optional&&) = default;
    ...
```

Conditionally deleted special members

Then you "simply" repeat this for each special member:

```
template<typename T>
  struct optional
  : maybe_copyable<is_copy_constructible_v<T>>,
    maybe_movable<is_move_constructible_v<T>>,
    maybe_copy_assignable<is_copy_assignable_v<T>>,
    maybe_move_assignable<is_move_assignable_v<T>>
  {
    optional(const optional&) = default;
    optional(optional&&) = default;
    ...
```

Conditionally explicit constructors

Similar to the last topic, a generic wrapper wants to preserve the "explicit-ness" of any converting constructors.

Otherwise you can get unsafe conversions to the wrapper where you don't want them:

```
void sink(unique_ptr<X>);
void bath(pair<unique_ptr<X>, int>);
X x;
sink(&x); // Won't compile, explicit constructor
bath( { &x, 1 } ); // uh-oh!
```

Conditionally explicit constructors

You don't simply want to make *all* converting constructors `explicit`.

It's safe (and convenient) for `{1, 2}` to convert to `pair<int, int>`.

So how do you make a constructor conditionally `explicit`, depending on whether the member it's constructing is `explicit`?

```
template<typename T1, typename T2>
  struct pair {
    template<typename U1, typename U2>
      EXPLICIT pair(U1&&, U2&&);
```

Conditionally explicit constructors

Define *two* constructors, one `explicit` and one not, and use SFINAE so at most one is enabled:

```
template<typename T1, typename T2>
  struct pair {
    // Can only be constructed explicitly:
    template<typename U1, typename U2,
        enable_if_t</*???*/, bool> = false>
      explicit pair(U1&&, U2&&);

    // Allows implicit conversions:
    template<typename U1, typename U2,
        enable_if_t</*???*/, bool> = false>
      pair(U1&&, U2&&);
```

Conditionally explicit constructors

```
// Can only be constructed explicitly:
template<typename U1, typename U2, enable_if_t<
          is_constructible<T1, U1>
       && is_constructible<T2, U2>
       && !(is_convertible<U1, U1>
            && is_convertible<U2, T2>), bool>=false>
  explicit pair(U1&&, U2&&);

// Allows implicit conversions:
template<typename U1, typename U2, enable_if_t<
          is_constructible<T1, U1>
       && is_constructible<T2, U2>
       && is_convertible<U1, T1>
       && is_convertible<U2, T2>, bool>=false>
  pair(U1&&, U2&&);
```

Conditionally explicit constructors

This might get slightly easier in C++2a:

```
template<typename U1, typename U2, enable_if_t<
        is_constructible<T1, U1>
     && is_constructible<T2, U2>, bool>=false>
  explicit(is_convertible<U1, T1>
           && is_convertible<U2, T2>)
  pair(U1&&, U2&&);
```

Using unique_ptr for exception safety

Somtimes, you have to perform two operations, either one of which might fail. This leads to complicated error code.

```
do_something(new int (23), new int (34));
```

Using unique_ptr for exception safety (2)

This is safe, but awkward.
Three cases is *very* awkward.

```
int *p1 = new int(23);
try {
    int *p2 = new int (34);
    do_something(p1, p2);
    }
catch (std::bad_alloc &) {
    delete p1;
    throw ;
    }
```

Using unique_ptr for exception safety (2)

unique_ptr makes writing exception-safe code easier.

```
std::unique_ptr<int> up1(new int(23));
std::unique_ptr<int> up2(new int(34));
do_something(up1.release(), up2.release());
```

See https://cplusplusmusings.wordpress.com/2015/03/09/simplifying-code-and-achieving-exception-safety-using-unique_ptr for a real-world example.

allocator_construct

Allocators have (optional) calls to construct and destroy objects.

```
template <typename T>
struct Alloc {
    typedef T value_type;
    T* allocate (size_t sz, const T* = 0);
    void deallocate (T *, size_t);

    template <class U, class... Args>
        void construct(U* p, Args&&... args) {
            ::new ((void*)p)
                U(std::forward<Args>(args)...);
        }

    void destroy(T *p) {p->~T();}
};
```

allocator_construct (2)

You can provide your own implementation of `construct` and `destroy`.

1. You can log object creation and destruction.
2. You can register and deregister objects in some global registry.
3. You can insert/ignore/change/rearrange constructor arguments.
4. You can decide what the "default value" is for the objects created by the allocator.

allocator_construct (3)

```
    vector<int, allocator<int>>    v1(5);
    vector<int, my_allocator<int>> v2(5);
    for (int i : v1) cout << i << ' ';
    cout << endl;
//  prints: 0 0 0 0 0
    for (int i : v2) cout << i << ' ';
    cout << endl;
//  prints: 3 3 3 3 3
```

tag dispatch

When implementing an algorithm, there are often different approaches you can take, depending on the characteristics of the data that you have to process.

There are different kinds of iterators: Input, Forward, Bidirectional and Random access.

tag dispatch (2)

The algorithm `find_end` takes two pairs of iterators, and returns the start of the last occurrence of the of the pattern in the corpus.

The most general implementation is to search for the pattern repeatedly until it fails, and then return the result of the last successful search.
If both the pattern and corpus are represented by bidirectional iterators (or better), then you can run the search backwards, and look for the *f*irst occurrence.
If both the pattern and corpus are represented by random-access iterators, then you can limit the range of the search even further.

tag dispatch (3)

```
template <typename Iter>
Iter algo(Iter first, Iter last,
          random_access_iterator_tag)
{ return first; }

template <typename Iter>
Iter algo(Iter first, Iter last,
          input_iterator_tag)
{ return last; }

template <typename Iter>
Iter algo(Iter first, Iter last) {
   return algo (first, last, typename
      iterator_traits<Iter>::iterator_category());
}
```

tag dispatch (4)

```
template <typename Iter>
Iter algo2(Iter first, Iter last) {
    if constexpr(is_same_v<
          iterator_traits<Iter>::iterator_category,
          random_access_iterator_tag>)
        return first;
    else
        return last;
}
```

Questions?

Thank you