Background

When calling an overloaded function, multiple functions might match the arguments provided in the call.

These functions are the viable candidates, and they form an overload set.

The process for finding the best function in this overload set is called **overload resolution**.

Partial ordering is a step in this process.

Overload Resolution

The objective of **overload resolution** is to find the single **best function** in the **overload set**.

This is the function which is going to be called.

If there is no single best function, then the call is diagnosed as **ambiguous**.

Overload Resolution

A quick overview of the steps involved:

- If there is an exact match, this is always the best candidate.
- Different kinds of conversions are ranked.
- Non-templates are better than templates.
- Certain kinds of references match other kinds better.

Overload Resolution

So far, this is looking at which function better matches the arguments provided to the call.

When a candidate is a function template, then function template specializations are generated, deduced from the arguments to the call.

Overload Resolution

These function template specializations participate in overload resolution just like any other non-template function.

This creates further challenge to the process.

Example 1

```
template <class T> void f(T);  // #1
template <class U> void f(U*); // #2
void g(int *a) {
  f(a);
}
```

For the call to f, there are two candidates:

1. `void f(int*)` specialization of #1
2. `void f(int*)` specialization of #2

Partial Ordering

Partial ordering is the last step of overload resolution, and is a tie breaker before we give up and diagnose the call as ambiguous.

This applies if the best matches so far are function template specializations.

Partial Ordering

Unlike the rest of overload resolution, this is not looking at the arguments of the call anymore.

This is also not looking at the function template specializations themselves, but at the function templates which they were generated from.

The objective of partial ordering is to find the function template which is more specialized.

More Specialized

```cpp
template <class T> void f(T);  // #1
template <class U> void f(U*); // #2
```

We need an intuition here: which function template looks more specialized?

- `#1` accepts any type.
- `#2` accepts any pointer.

More Specialized

A pointer is a type, but not all types are pointers.

Intuitively, something more specialized works better in a more narrow set of circumstances.

This looks to apply to `#2`.

More Specialized

Partial ordering works by trying to figure out if there is a template function which strictly accepts fewer types of arguments than all of the others.

Given a function A, does it accept the stuff which B accepts, and is the reverse not accepted?

Example

```
template <class T> void f(T);  // #1
template <class U> void f(U*); // #2
```

The argument synthesized from the parameter of `#2` is of type `U*`.

As the first step after calling `#1` with that argument,
we need to perform template argument deduction.

Template Argument Deduction

We are trying to balance the following equation:

```
T = U*
```

Where the left-hand side is the parameter, and the right hand side is the argument.

We want to figure out what should be the type of `T`, such that `T` equals `U*`.

Template Argument Deduction

Does `#2` accept the stuff which `#1` accepts?

For this case, we have `U* = T`, where we are trying to deduce `U`.

There is no possible type we could deduce `U` as, such that `U*` would equal `T`.

More Specialized

So `#1` accepts the stuff which `#2` accepts, but `#2` does not accept the stuff which `#1` accepts.

This answers the full question: `#2` is indeed more specialized, which means `#2` is going to be called in the original example.

Class Template Partial Specialization

The rules for function template partial ordering are also used to determine which class template partial specialization gets used.

```
template <class T> struct A     {}; // #1
template <class U> struct A<U*> {}; // #2
template struct A<void*>;
```

Class Template Partial Specialization

According to the rules in the standard, this works by rewriting the class template candidates into function templates in a certain way and proceeding as if **partial ordering** these candidates.

Class Template Partial Specialization

Each candidate is rewritten into a function templates taking one argument, which is its partial specialization signature.

```
template <class U> struct A<U*> {}; // #2
```

Becomes:

```
template <class U> void f(A<U*>);
```

The Primary Template

For this process, the primary template is equivalent to a partial specialization which specializes no parameters.

```
template <class T> struct A {}; // #1
```

Translates to:

```
template <class T> struct A<T> {}; // #1
```

The Primary Template

Which is not a valid partial specialization, as it specializes no arguments, but conceptually it follows that the rewritten candidate will be:

```
template <class T> void f(A<T>);
```

Specializations are more specialized

As a first step, we must determine that the partial specialization must be more specialized than the primary template.

If it's not, the program is ill-formed before we even get to the last line.

Example 2 as rewritten

```
template <class T> void f(A<T>);  // #1
template <class U> void f(A<U*>); // #2
```

We can deduce `T = U*` to make `A<T> = A<U*>` hold true,
but we can find no such deduction to make the opposite work.

Specializations are more specialized

Back to example 2, we don't need to check which is more specialized again. If the instantiation matches at least one partial specialization, we don't need to insert the primary template in the candidate set.

```
template <class T> struct A      {}; // #1
template <class U> struct A<U*> {}; // #2
template struct A<void*>;
```

Template argument deduction

The rules of template argument deduction have limitations.

This plays a major role in how effective partial ordering can be.

They can't magically find the types for the template parameters which would satisfy the equations as presented.

Template argument deduction

It works through a simple process of pattern matching,
and a final pass where the template parameters which were deduced are substituted in the parameter, and confirm whether it matches the argument.

Pattern matching on types

Given template parameter `T`, a parameter `P` and an argument `A`:

```
P = T(*)(T)
A = U(*)(V)
```

Where T must be deduced, this starts a recursive descent into both types at the same time.

- Both are pointers, this matches, so it continues looking into the pointees.

  ```
  P = T(T)
  A = U(V)
  ```

Pattern matching on types

Both are function types, this matches, so at this point we fork into two sets of `P` and `A`, one for the return type, the other for the single parameter type.

For the return type:

```
P = T
A = U
```

- This deduces `T = U`.

Pattern matching on types

For the parameter type:

```
P = T
A = V
```

- This tries to deduce that `T = V`, but `T` was deduced as `U` earlier. This is an inconsistent deduction and we can stop here.

Non-deduced contexts

This recursive descent into types has one crucial limitation:
It can't look into certain constructs, like type lookup
into a dependent type.

Given:

`P = typename T::type`

`A = U`

Here no template parameters can be deduced.

Non-deduced contexts

This holds true even for certain trivial standard constructs, like `std::type_identity<T>`: deduction cannot "see" that its member `type` equals to `T`.

This can't be changed either, as user code relies on this assumption.

At the end of the process when we compare the substituted `P` with `A`, we will figure out if this was a problem or not.

Non-deduced mismatch

Is the partial specialization #2 more specialized than the primary template #1?

```
template<class T, class U> struct X {}; // #1

template<class V>
struct X<V, typename V::type> {}; // #2
```

Non-deduced mismatch

Lets rewrite this into function template partial ordering:

```
template<class T, class U> void f(X<T, U>);              // #1
template<class V>          void f(X<V, typename V::type>); // #2
```

Does `#1` accept the stuff which `#2` accepts?

Lets call `#1`, where the argument is the parameter of `#2`.

`P = X<T, U>`
`A = X<V, typename V::type>`

- Both are template specializations of `X`, so this matches and we move on to their arguments.
  1. `P = T`
     `A = V`
     - We deduce `T = V`.
  2. `P = U`
     `A = typename V::type`
     - We deduce `U = typename V::type`.

Non-deduced mismatch

Now lets substitute our deduction into P:
`P' = X<V, typename V::type>`.

This is the same as `A` we started with, so this is a match.

So `#1` does accept the stuff which `#2` accepts.

But this only answers half the question.

Non-deduced mismatch

```
P = X<V, typename V::type>
A = X<T, U>
```

Both are template specializations of `X`, move on as before.

-
  ```
  P = V
  A = T
  ```
  - We deduce `V = T`.

Non-deduced mismatch

```
P = typename V::type
```

```
A = U
```

- We can't see through the dependent type lookup, so forget we ever saw this and move on.

Non-deduced mismatch

Now lets substitute our deduction into P: `P' = X<T, typename T::type>`.

This does not match the `A` we started with, so this fails.

Since `#1` accepts the stuff which `#2` accepts, but the reverse does not hold true, this implies `#2` is indeed more specialized.

Function vs Class

Back to the last example: https://compiler-explorer.com/z/rbvsP7ce4

```
template<class T, class U> struct X {}; // #1

template<class V>
struct X<V, typename V::type> {}; // #2
```

Every implementation agrees this is valid, so good so far!

Function vs Class

https://compiler-explorer.com/z/rzEracEdo

```
template <class T, class U> struct X {};

template<class T, class U> void f(X<T, U>);
template<class V>          void f(X<V, typename V::type>) {}

struct Y {
  using type = int;
};

void g(X<Y, int> x) {
  f(x);
}
```

CWG2160

Turns out, just performing the check in that case is not so simple.

It breaks a test case like: https://compiler-explorer.com/z/Ex5bndoY1

```cpp
template<class T> struct identity { using type = T; };
template<class U> void f(U, typename identity<U>::type);
template<class V> void f(V, int);
void g() {
  f(0, 0);
}
```

Bonus

Last example: https://compiler-explorer.com/z/c75WbzMhh

```
template <class T1, class T2> struct X {};

template<class U1, class U2> void f(X<U1, U2>);       // #1
template<class V1> void f(X<V1, typename V1::type>); // #2
template<class V2> void f(X<V2, V2>);                // #3

struct Y { using type = Y; };

void g(X<Y, Y> x) {
  f(x);
}
```

Bonus

Let's rewrite this back to class template partial specialization: https://compiler-explorer.com/z/oYT1ovofT

```
template <class T1, class T2> struct X {};                    // #1
template <class V1> struct X<V1, typename V1::type> {}; // #2
template <class V2> struct X<V2, V2> {};                // #3

struct Y {
  using type = Y;
};

template struct X<Y, Y>;
```

Conclusion

Understanding where partial ordering is situated in overload resolution is important so as not to get confused when the result of overload resolution was influenced by it or not.

There is substantial implementation divergence for functions, but not so much for class template partial specializations.

For experiments, writing test cases in terms of class template partial specialization is much easier.