Back To Basics
Name Lookup and Overload Resolution

MATEUSZ PUSZ

Cppcon
The C++ Conference
2022
September 12th-16th

Workshopy Style

Workshopy Style

- Provide rationale
- Facilitate discussion
  - force the audience to think
  - not just a lecture
- Describe
  - pitfalls
  - corner cases
- Provide recommendations
- ~~Lot's of coding~~

https://ahaslides.com/NLOR

Name Lookup and Overload Resolution

Name Lookup and Overload Resolution are among the most complex and the most expensive at compile time features of the C++ language.

Why to allow more than one function with the same name?

Why to allow more than one function with the same name?

> Overload Set is the "atom" of C++ API design.
>
> *-- Titus Winters*

Why to allow more than one function with the same name?

> Overload Set is the "atom" of C++ API design.
>
> *-- Titus Winters*

TERSE, ROBUST, AND FAST INTERFACES

```
void print(const X& x);
void print(const Y& y);
```

```
void print_X(const X& x);
void print_Y(const Y& y);
```

Why to allow more than one function with the same name?

Overload Set is the "atom" of C++ API design.

-- Titus Winters

GENERIC PROGRAMMING

- No dynamic polymorphism needed to have a single interface entry point

```
void print(const auto& v);
```

```
void print(const printable& v);
```

Why to allow more than one function with the same name?

> Overload Set is the "atom" of C++ API design.
>
> *-- Titus Winters*

FRIENDLY SYNTAX WITH OPERATORS OVERLOADING

```
MyInt a{1}, b{2};
auto res = a + b;
```

```
MyInt a{1}, b{2};
auto res = add(a, b);
```

Why to allow more than one function with the same name?

> Overload Set is the "atom" of C++ API design.
>
> *-- Titus Winters*

CUSTOMIZATION POINTS

```
struct X { int value; };
std::ostream& operator<<(std::ostream& os, const X& x);
```

```
X x;
std::cout << x << "\n";
```

Ad hoc polymorphism

Function has different implementations depending on a *limited range of individually specified types and combinations*.

```
void print(const X& x);
void print(const Y& y);
```

Ad hoc polymorphism

Function has different implementations depending on a *limited range of individually specified types and combinations*.

```
void print(const X& x);
void print(const Y& y);
```

In the C++ language multiple functions and function templates **may share the same name**. Each of them **must have a different set of parameters or template parameter constraints**, and may provide different return types. The compiler will **select the best matching function at compile-time**.

Overload sets

C.162: Overload operations that are roughly equivalent.

-- C++ Core Guidelines

GOOD

```
void print(int a);
void print(int a, int base);
void print(const string&);
```

BAD

```
void print_int(int a);
void print_based(int a, int base);
void print_string(const string&);
```

Overload sets

C.163: Overload only for operations that are roughly equivalent.

-- C++ Core Guidelines

GOOD

```
void open_gate(Gate& g);    // remove obstacle from garage exit lane
void fopen(const char* name, const char* mode);    // open file
```

BAD

```
void open(Gate& g);    // remove obstacle from garage exit lane
void open(const char* name, const char* mode ="r");    // open file
```

Overload sets

Use overloaded functions (including constructors) only if a reader looking at a call site **can get a good idea of what is happening without having to first figure out exactly which overload is being called**.

-- Google C++ Style

Properties of a good overload set

- Correctness can be judged at the call site **without knowing which overload is picked**
- A single comment can **describe the full set**
- Each element of the set is **doing "the same thing"**

Properties of a good overload set

- Correctness can be judged at the call site **without knowing which overload is picked**
- A single comment can **describe the full set**
- Each element of the set is **doing "the same thing"**

EXAMPLE

```
void vector<T>::push_back(const T&);
void vector<T>::push_back(T&&);
```

```
v.push_back("hello"s);
v.push_back(std::move(world));
```

- If we remove the second overload, the same behavior but not performance

A process of calling a function in C++

A process of calling a function in C++

1 Name Lookup

A process of calling a function in C++

1. Name Lookup
2. Template Argument Deduction

A process of calling a function in C++

1. Name Lookup
2. Template Argument Deduction
3. Overload Resolution

A process of calling a function in C++

1. Name Lookup
2. Template Argument Deduction
3. Overload Resolution
4. Member Access Rules

A process of calling a function in C++

1. Name Lookup
2. Template Argument Deduction
3. Overload Resolution
4. Member Access Rules
5. Function Template Specializations

A process of calling a function in C++

1. Name Lookup
2. Template Argument Deduction
3. Overload Resolution
4. Member Access Rules
5. Function Template Specializations
6. Virtual Dispatch

A process of calling a function in C++

1. Name Lookup
2. Template Argument Deduction
3. Overload Resolution
4. Member Access Rules
5. Function Template Specializations
6. Virtual Dispatch
7. Deleting Functions

A process of calling a function in C++

1. Name Lookup
2. Template Argument Deduction
3. Overload Resolution
4. Member Access Rules
5. Function Template Specializations
6. Virtual Dispatch
7. Deleting Functions

Name Lookup

Name lookup is the procedure by which a name, when encountered in a program, is associated with the declaration that introduced it.

Name Lookup

Name lookup is the procedure by which a name, when encountered in a program, is associated with the declaration that introduced it.

The result of the function name lookup process is a set of candidate functions.

Name Lookup

QUALIFIED NAME LOOKUP

UNQUALIFIED NAME LOOKUP

Name Lookup

QUALIFIED NAME LOOKUP

- Name that appears *on the right hand side* of the scope resolution operator `::`

UNQUALIFIED NAME LOOKUP

- Name that does *not appear to the right* of a scope resolution operator `::`

Name Lookup

QUALIFIED NAME LOOKUP

- Name that appears *on the right hand side* of the scope resolution operator `::`

UNQUALIFIED NAME LOOKUP

- Name that does *not appear to the right* of a scope resolution operator `::`

Before a qualified name lookup can be performed for the name on the right hand side of `::`, lookup must be completed for the name on its left hand side.

Name Lookup

QUALIFIED NAME LOOKUP

- Name that appears *on the right hand side* of the scope resolution operator **::**

UNQUALIFIED NAME LOOKUP

- Name that does *not appear to the right* of a scope resolution operator **::**

```cpp
#include <iostream>

int main()
{
  struct std {};
//std::cout << "fail\n"; // Error: unqualified lookup for 'std' finds the struct
  ::std::cout << "ok\n"; // OK: ::std finds the namespace std
}
```

- **cout** above is always looked up in a qualified way

Qualified Name Lookup

- A qualified name **may refer to**
  - class member
  - namespace member
  - enumerator

Qualified Name Lookup

- A qualified name may refer to
  - class member
  - namespace member
  - enumerator

If there is nothing on the left hand side of the `::`, the lookup considers only declarations made in the global namespace scope or introduced into the global namespace by a `using` declaration.

AhaSlides: Which function is selected?

```cpp
namespace my_namespace {
void func(const std::string&);

namespace internal {
void func(int);

namespace deep {
void test()
{
  std::string s("hello");
  func(s);
}

} // namespace deep
} // namespace internal
} // namespace my_namespace
```

AhaSlides: Which function is selected?

```
namespace my_namespace {
void func(const std::string&);

namespace internal {
void func(int);

namespace deep {
void test()
{
  std::string s("hello");
  func(s);
}

} // namespace deep
} // namespace internal
} // namespace my_namespace
```

```
error: cannot convert 'std::string' to 'int' for argument '1' to
       'void my_namespace::internal::func(int)'
```

Unqualified Name Lookup

Unqualified name lookup examines the scopes, **until it finds at least one declaration of any kind** (does not have to be a function) **with a matching name**, at which time **scopes traversing stops** and no further scopes are examined.

Unqualified Name Lookup

Unqualified name lookup examines the scopes, **until it finds at least one declaration of any kind** (does not have to be a function) **with a matching name**, at which time **scopes traversing stops** and no further scopes are examined.

Even if there are other functions with the same name in the outer scopes, they are hidden from the lookup process.

Nested namespaces: Pitfalls

```
namespace my_namespace {
void func(double);

namespace internal {

namespace deep {
void test()
{
  func(3.14);
}

} // namespace deep
} // namespace internal
} // namespace my_namespace
```

Nested namespaces: Pitfalls

```
namespace my_namespace {
void func(double);

namespace internal {
void func(int);

namespace deep {
void test()
{
  func(3.14);
}

} // namespace deep
} // namespace internal
} // namespace my_namespace
```

Program still compiles fine but may not work as expected.

Recommendation: Nested namespaces

To prevent unexpected lookup problems try to keep your namespaces flat and shallow wherever possible.

Recommendation: Nested namespaces

To prevent unexpected lookup problems try to keep your namespaces flat and shallow wherever possible.

This is C++, not Java ;-)

Nested overloads

The same scoping-based lookup rules apply to overloads in a class hierarchy.

Nested overloads

The same scoping-based lookup rules apply to overloads in a class hierarchy.

```
struct Base {
  void func(double);
};

struct Derived : Base {
  void func(int);
};
```

```
Derived d;
d.func(3.14);
```

Name Lookup from regular functions

```
void func(int);
void func(double);

namespace N1 {

void test()
{
  std::string s("hello");
  func(s);
}

void func(const std::string&);

}
```

```
N1::test();
```

Name Lookup from regular functions

```
void func(int);
void func(double);

namespace N1 {

void test()
{
  std::string s("hello");
  func(s);
}

void func(const std::string&);

}
```

```
N1::test();
```

```
error: no matching function for call to 'func(std::string&)'
note: candidate: 'void func(int)'
note:   no known conversion for argument 1 from 'std::string' to 'int'
note: candidate: 'void func(double)'
note:   no known conversion for argument 1 from 'std::string' to 'double'
```

Name Lookup from regular functions

```cpp
void func(int);
void func(double);

namespace N1 {

void test()
{
  std::string s("hello");
  func(s);
}

void func(const std::string&);

}
```

```cpp
N1::test();
```

Name lookup in regular function considers only function candidates *visible from its definition context*.

Argument-dependent lookup (ADL, Koenig lookup)

```
namespace N2 {

struct X {};
void func(const X&);

}

namespace N1 {

void test(N2::X x) { func(x); }

}
```

```
N2::X x{};
N2::func(x);
N1::test(x);
```

Argument-dependent lookup (ADL, Koenig lookup)

```
namespace N2 {

struct X {};
void func(const X&);

}

namespace N1 {

void test(N2::X x) { func(x); }

}
```

```
N2::X x{};
N2::func(x);
N1::test(x);
```

ADL is the set of rules for looking up the unqualified function names in function-call expressions. These function names are *looked up in the namespaces of their arguments*.

AhaSlides: Tell me about the code?

```cpp
namespace a {
  using my_array = std::vector<int>;

  void print(const my_array& array)
  {
    // ...
  }
}

void foo(const a::my_array& array)
{
  print(array);
}
```

```cpp
namespace a {
  struct my_array { std::vector<int> data; };

  void print(const my_array& array)
  {
    // ...
  }
}

void foo(const a::my_array& array)
{
  print(array);
}
```

AhaSlides: Tell me about the code?

```cpp
namespace a {
  using my_array = std::vector<int>;

  void print(const my_array& array)
  {
    // ...
  }
}

void foo(const a::my_array& array)
{
  print(array);
}
```

```
error: 'print' was not declared in this scope
```

```cpp
namespace a {
  struct my_array { std::vector<int> data; };

  void print(const my_array& array)
  {
    // ...
  }
}

void foo(const a::my_array& array)
{
  print(array);
}
```

```
Compiler returned: 0
```

AhaSlides: Tell me about the code?

```cpp
namespace a {
  using my_array = std::vector<int>;

  void print(const my_array& array)
  {
    // ...
  }
}

void foo(const a::my_array& array)
{
  print(array);
}
```

```
error: 'print' was not declared in this scope
```

```cpp
namespace a {
  struct my_array { std::vector<int> data; };

  void print(const my_array& array)
  {
    // ...
  }
}

void foo(const a::my_array& array)
{
  print(array);
}
```

```
Compiler returned: 0
```

The aliases are fully resolved and expanded to their source types before the list of namespaces to ADL search are chosen.

AhaSlides: Tell me about the code?

```cpp
namespace N2 {

struct X {
  friend void func(const X&) { /* ... */ }
};

}

namespace N1 {

void test(N2::X x) { func(x); }  // OK

}
```

```cpp
N2::X x{};
N2::func(x);    // ???
N1::test(x);    // ???
```

AhaSlides: Tell me about the code?

```cpp
namespace N2 {

struct X {
  friend void func(const X&) { /* ... */ }
};

}

namespace N1 {

void test(N2::X x) { func(x); }  // OK

}
```

```cpp
N2::X x{};
N2::func(x);   // ???
N1::test(x);   // ???
```

```
error: 'func' is not a member of 'N2'
```

Hidden Friends

Friend function *publicly declared and defined inside* of a class and *taking this class type as an argument* is called a **Hidden Friend**.

```
struct X {
  friend void func(const X&) { /* ... */ }
};
```

Hidden Friends

Friend function *publicly declared and defined inside* of a class and *taking this class type as an argument* is called a **Hidden Friend**.

```
struct X {
  friend void func(const X&) { /* ... */ }
};
```

Such function **can be found only through the ADL**.

Recommendation: Hidden Friends

Prefer Hidden Friend functions rather than global non-member functions **to overload operators or implement other common customization points**. *Do it even when access to the private class members is not required* in the function's definition.

Recommendation: Hidden Friends

Prefer Hidden Friend functions rather than global non-member functions **to overload operators or implement other common customization points**. *Do it even when access to the private class members is not required* in the function's definition.

`friend` functions are not a part of the candidate set for arguments of other types which means they make the name lookup and overload resolution process faster.

Name Lookup from function templates

```cpp
namespace N1 {

void test(auto v)  // look Ma, a template!
{
  func(v);     // OK
  func(123);   // Error
}

void func(int);   // not found

}

namespace N2 {

struct X {};
void func(const X&);   // found

}
```

```cpp
N2::X x{};
N1::test(x);
```

```
error: there are no arguments to 'func' that depend on a template parameter,
       so a declaration of 'func' must be available
```

Name Lookup from function templates

```
namespace N1 {

void test(auto v)
{
  func(v);     // Error
}

}

namespace N2 {

struct X {};

}
```

```
void func(const N2::X&);  // not found

N2::X x{};
N1::test(x);
```

```
error: 'func' was not declared in this scope, and no declarations were found by
       argument-dependent lookup at the point of instantiation
```

Dependent names

- Inside the definition of a template, the *meaning of some constructs may differ* from one instantiation to another

Dependent names

- Inside the definition of a template, the *meaning of some constructs may differ* from one instantiation to another
- **Types and expressions may depend on types and values of template parameters**

```
template<typename T>
struct X : B<T>              // B<T> is dependent on T
{
  typename T::A* pa;         // T::A is dependent on T
  void f(B<T>* pb)
  {
    static int i = B<T>::i; // B<T>::i is dependent on T
    pb->j++;                 // pb->j is dependent on T
  }
};
```

Dependent names

- Inside the definition of a template, the *meaning of some constructs may differ* from one instantiation to another
- **Types and expressions may depend on types and values of template parameters**

```
template<typename T>
struct X : B<T>              // B<T> is dependent on T
{
  typename T::A* pa;         // T::A is dependent on T
  void f(B<T>* pb)
  {
    static int i = B<T>::i; // B<T>::i is dependent on T
    pb->j++;                 // pb->j is dependent on T
  }
};
```

- *Name lookup* and *binding are different* for dependent names and non-dependent names

Dependent names: Binding

- *Non-dependent* names are looked up and bound *at the point of template definition*

Dependent names: Binding

- *Non-dependent* names are looked up and bound *at the point of template definition*
- This binding *holds even if* at the point of template instantiation *there is a better match*

```
void g(double);

template<class T>
struct S {
  void f() const { g(1); } // "g" is a non-dependent name, bound now
};

void g(int);
```

```
g(1);        // calls g(int)
S<int> s;
s.f();       // calls g(double)
```

Dependent names: Binding

- *Non-dependent* names are looked up and bound *at the point of template definition*
- This binding *holds even if* at the point of template instantiation *there is a better match*

```
void g(double);

template<class T>
struct S {
  void f() const { g(1); } // "g" is a non-dependent name, bound now
};

void g(int);
```

```
g(1);        // calls g(int)
S<int> s;
s.f();       // calls g(double)
```

- Binding of *dependent names* is postponed *until lookup takes place*

Dependent names: Lookup

- The lookup of a dependent name used in a template is *postponed until the template arguments are known*

Dependent names: Lookup

- The lookup of a dependent name used in a template is *postponed until the template arguments are known*
- **non-ADL lookup** examines function declarations with external linkage that are *visible from the template definition context*

Dependent names: Lookup

- The lookup of a dependent name used in a template is *postponed until the template arguments are known*
- **non-ADL lookup** examines function declarations with external linkage that are *visible from the template definition context*
- **ADL** examines function declarations with external linkage that are *visible from both the template definition* context and the *template instantiation* context

Dependent names: Lookup

- The lookup of a dependent name used in a template is *postponed until the template arguments are known*
- **non-ADL lookup** examines function declarations with external linkage that are *visible from the template definition context*
- **ADL** examines function declarations with external linkage that are *visible from both the template definition* context and the *template instantiation* context
- Adding a new function declaration *after template definition does not make it visible, except via ADL*

Generic frameworks with customization points

convert.h

```
template<typename T>
void convert(std::string_view str, T& out)
{
  // default implementation
}

template<typename T>
T from_string(std::string_view str)
{
  T t;
  convert(str, t);
  return t;
}
```

Generic frameworks with customization points

convert.h

```
template<typename T>
void convert(std::string_view str, T& out)
{
  // default implementation
}

template<typename T>
T from_string(std::string_view str)
{
  T t;
  convert(str, t);
  return t;
}
```

price.h

```
struct price { int value; };

void convert(std::string_view str, price& p)
{
  convert(str, p.value);
}
```

Generic frameworks with customization points

convert.h

```
template<typename T>
void convert(std::string_view str, T& out)
{
  // default implementation
}

template<typename T>
T from_string(std::string_view str)
{
  T t;
  convert(str, t);
  return t;
}
```

price.h

```
struct price { int value; };

void convert(std::string_view str, price& p)
{
  convert(str, p.value);
}
```

main.cpp

```
#include "convert.h"
#include "price.h"

price p = from_string<price>("123");
```

Generic frameworks with customization points

convert.h

```cpp
template<typename T>
void convert(std::string_view str, T& out)
{
  // default implementation
}

template<typename T>
T from_string(std::string_view str)
{
  T t;
  convert(str, t);
  return t;
}
```

price.h

```cpp
struct price { int value; };

void convert(std::string_view str, price& p)
{
  convert(str, p.value);
}
```

main.cpp

```cpp
#include "convert.h"
#include "price.h"

price p = from_string<price>("123");
```

All customization points benefit from this property.

AhaSlides: swap() the type

```
template<typename T>
struct wrapper {
  T data_;
};

template<typename T>
void swap(wrapper<T>& lhs,
          wrapper<T>& rhs)
  noexcept(std::is_nothrow_swappable_v<T>)
{
  // ???
}
```

```
wrapper<std::string> s1, s2;
swap(s1, s2);
```

Which of the following will work correctly?

- `swap(lhs.data_, rhs.data_);`
- `std::swap(lhs.data_, rhs.data_);`
- `std::ranges::swap(lhs.data_, rhs.data_);`

Answer: swap() the type

```
template<typename T>
struct wrapper {
  T data_;
};

template<typename T>
void swap(wrapper<T>& lhs,
          wrapper<T>& rhs)
  noexcept(std::is_nothrow_swappable_v<T>)
{
  // ???
}
```

```
wrapper<std::string> s1, s2;
swap(s1, s2);
```

All work as expected

AhaSlides: swap() the type

```
template<typename T>
struct wrapper {
  T data_;
};

template<typename T>
void swap(wrapper<T>& lhs,
          wrapper<T>& rhs)
  noexcept(std::is_nothrow_swappable_v<T>)
{
  // ???
}
```

```
struct X {};

void swap(X&, X&)
{ std::cout << "my swap\n"; }
```

```
wrapper<std::string> s1, s2;
swap(s1, s2);

wrapper<X> x1, x2;
swap(x1, x2);
```

Which of the following will work correctly?

- `swap(lhs.data_, rhs.data_);`
- `std::swap(lhs.data_, rhs.data_);`
- `std::ranges::swap(lhs.data_, rhs.data_);`

Answer: swap() the type

```
template<typename T>
struct wrapper {
  T data_;
};

template<typename T>
void swap(wrapper<T>& lhs,
          wrapper<T>& rhs)
  noexcept(std::is_nothrow_swappable_v<T>)
{
  // ???
}
```

```
struct X {};

void swap(X&, X&)
{ std::cout << "my swap\n"; }
```

```
wrapper<std::string> s1, s2;
swap(s1, s2);

wrapper<X> x1, x2;
swap(x1, x2);
```

- **`std::swap()`** compiles but does not work as expected
- **`swap()`** and **`std::ranges::swap()`** work correctly

AhaSlides: swap() the type

```
template<typename T>
struct wrapper {
  T data_;
};

template<typename T>
void swap(wrapper<T>& lhs,
          wrapper<T>& rhs)
  noexcept(std::is_nothrow_swappable_v<T>)
{
  // ???
}
```

```
struct X {};

void swap(X&, X&)
{ std::cout << "my swap\n"; }
```

```
wrapper<std::string> s1, s2;
swap(s1, s2);

wrapper<X> x1, x2;
swap(x1, x2);

wrapper<int> i1, i2;
swap(i1, i2);
```

Which of the following will work correctly?

- `swap(lhs.data_, rhs.data_);`
- `std::swap(lhs.data_, rhs.data_);`
- `std::ranges::swap(lhs.data_, rhs.data_);`

Answer: swap() the type

```
template<typename T>
struct wrapper {
  T data_;
};

template<typename T>
void swap(wrapper<T>& lhs,
          wrapper<T>& rhs)
  noexcept(std::is_nothrow_swappable_v<T>)
{
  // ???
}
```

```
struct X {};

void swap(X&, X&)
{ std::cout << "my swap\n"; }
```

```
wrapper<std::string> s1, s2;
swap(s1, s2);

wrapper<X> x1, x2;
swap(x1, x2);

wrapper<int> i1, i2;
swap(i1, i2);
```

- **swap()** does not compile
- **std::swap()** does not work (but compiles)
- **std::ranges::swap()** works as expected

using declarations

```cpp
template<typename T>
struct wrapper {
  T data_;
};

template<typename T>
void swap(wrapper<T>& lhs,
          wrapper<T>& rhs)
  noexcept(std::is_nothrow_swappable_v<T>)
{
  using std::swap;
  swap(lhs.data_, rhs.data_);
}
```

```cpp
struct X {};

void swap(X&, X&)
{ std::cout << "my swap\n"; }
```

```cpp
wrapper<std::string> s1, s2;
swap(s1, s2);

wrapper<X> x1, x2;
swap(x1, x2);

wrapper<int> i1, i2;
swap(i1, i2);
```

Using declarations can be used to introduce namespace members into other namespaces and block scopes.

Customization Point Object (CPO, Niebloid)

```
template<typename T>
struct wrapper {
  T data_;
};

template<typename T>
void swap(wrapper<T>& lhs,
          wrapper<T>& rhs)
  noexcept(std::is_nothrow_swappable_v<T>)
{
  std::ranges::swap(lhs.data_, rhs.data_);
}
```

```
struct X {};

void swap(X&, X&)
{ std::cout << "my swap\n"; }
```

```
wrapper<std::string> s1, s2;
swap(s1, s2);

wrapper<X> x1, x2;
swap(x1, x2);

wrapper<int> i1, i2;
swap(i1, i2);
```

Customization Point Object (CPO, Niebloid)

```cpp
template<typename T>
struct wrapper {
  T data_;
};

template<typename T>
void swap(wrapper<T>& lhs,
          wrapper<T>& rhs)
  noexcept(std::is_nothrow_swappable_v<T>)
{
  std::ranges::swap(lhs.data_, rhs.data_);
}
```

```cpp
struct X {};

void swap(X&, X&)
{ std::cout << "my swap\n"; }
```

```cpp
wrapper<std::string> s1, s2;
swap(s1, s2);

wrapper<X> x1, x2;
swap(x1, x2);

wrapper<int> i1, i2;
swap(i1, i2);
```

- Customization interface *entry point can not be found through ADL*
- Customization point *function found only via ADL*

Customization Point Object (CPO, Niebloid)

```
namespace std::ranges {
  namespace swap_impl {









    struct fn {










    };
  }

  inline constexpr swap_impl::fn swap;
}
```

Customization Point Object (CPO, Niebloid)

```
namespace std::ranges {
  namespace swap_impl {



    template<typename T>
    inline constexpr bool has_customization =
      requires(T& t) {
        swap(t, t);  // uses ADL
      };

    struct fn {










    };
  }

  inline constexpr swap_impl::fn swap;
}
```

Customization Point Object (CPO, Niebloid)

```
namespace std::ranges {
  namespace swap_impl {



    template<typename T>
    inline constexpr bool has_customization =
      requires(T& t) {
        swap(t, t);  // uses ADL
      };

    struct fn {
      template<typename T>
      constexpr void operator()(T& lhs, T& rhs) const
      {
        if constexpr(has_customization<T>)
          swap(lhs, rhs);  // uses ADL
        else {
          // default implementation
          // ...
        }
      }
    };
  }

  inline constexpr swap_impl::fn swap;
}
```

Customization Point Object (CPO, Niebloid)

```
namespace std::ranges {
  namespace swap_impl {
    // non-ADL lookup block (poison pill)
    void swap(); // undefined

    template<typename T>
    inline constexpr bool has_customization =
      requires(T& t) {
        swap(t, t);  // uses ADL
      };

    struct fn {
      template<typename T>
      constexpr void operator()(T& lhs, T& rhs) const
      {
        if constexpr(has_customization<T>)
          swap(lhs, rhs);  // uses ADL
        else {
          // default implementation
          // ...
        }
      }
    };
  }

  inline constexpr swap_impl::fn swap;
}
```

Customization Point Object (CPO, Niebloid)

```
namespace std::ranges {
  namespace swap_impl {
    // non-ADL lookup block (poison pill)
    void swap(); // undefined

    template<typename T>
    inline constexpr bool has_customization =
      requires(T& t) {
        swap(t, t);  // uses ADL
      };

    struct fn {
      template<typename T>
      constexpr void operator()(T& lhs, T& rhs) const
      {
        if constexpr(has_customization<T>)
          swap(lhs, rhs);  // uses ADL
        else {
          // default implementation
          // ...
        }
      }
    };
  }

  inline constexpr swap_impl::fn swap;
}
```

```
struct X {};

void swap(X&, X&)
{ std::cout << "my swap\n"; }
```

```
std::string s1, s2;
std::ranges::swap(s1, s2);

X x1, x2;
std::ranges::swap(x1, x2);

int i1{1}, i2{2};
std::ranges::swap(i1, i2);
```

AhaSlides: Which is being called?

```
template<class T> void f(T);     // #1
template<>        void f(int*); // #2
template<class T> void f(T*);   // #3

f(new int{1});
```

Overloads vs explicit function template specializations

```
template<class T> void f(T);     // #1: overload for all types
template<>        void f(int*); // #2: specialization of #1 for pointers to int
template<class T> void f(T*);   // #3: overload for all pointer types

f(new int{1});   // calls #3, even though specialization of #1 would be a perfect match
```

- Only **non-template** and **primary template** overloads participate in *overload resolution*
- The **specializations** are not overloads and *are not considered*
- Only *after the overload resolution selects* the best-matching **primary function template**, its ***specializations*** *are examined* to see if one is a better match

A process of calling a function in C++

1. Name Lookup
2. Template Argument Deduction
3. **Overload Resolution**
4. Member Access Rules
5. Function Template Specializations
6. Virtual Dispatch
7. Deleting Functions

Overload Resolution

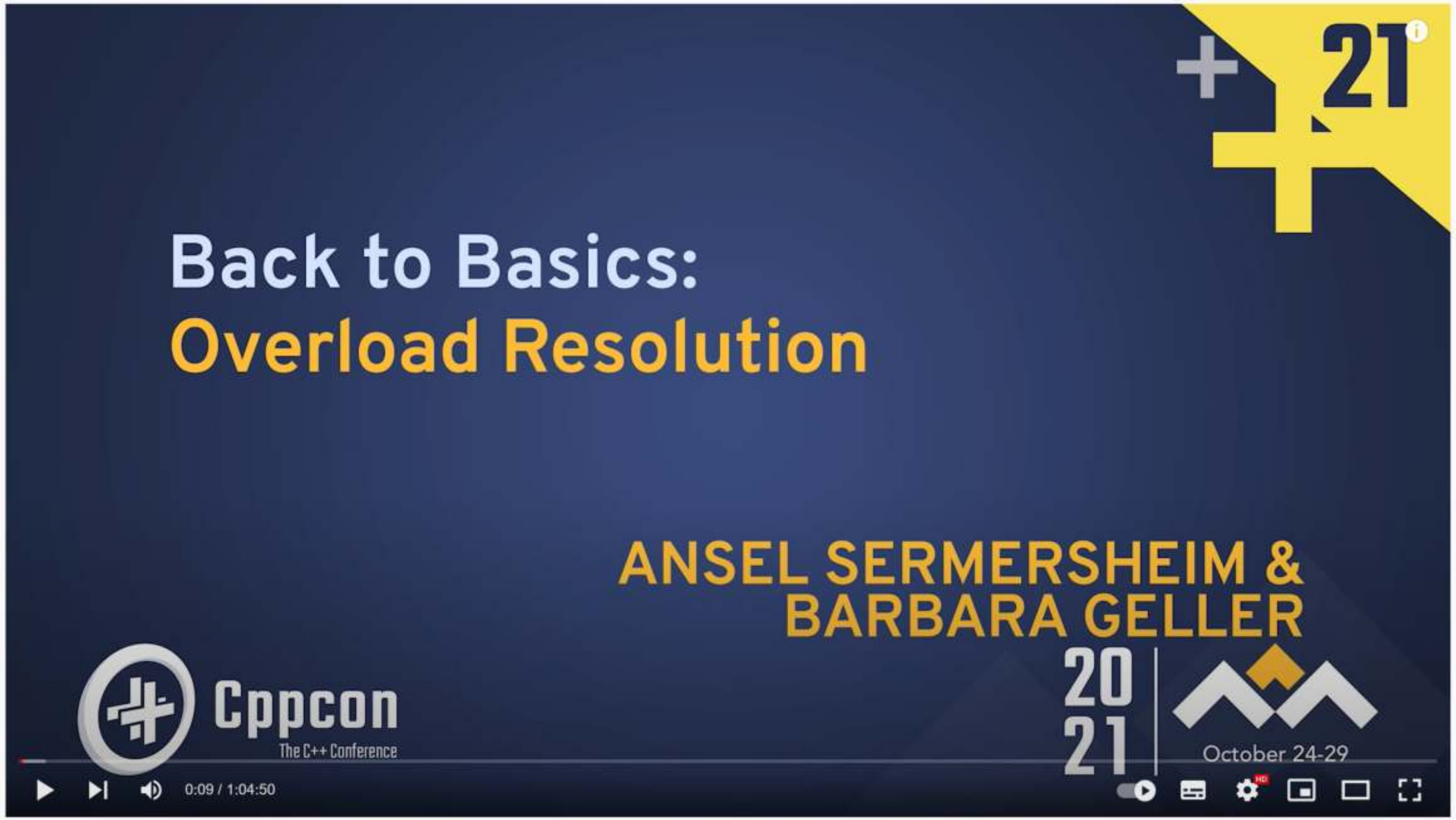
A process of **selecting the most appropriate overload** at compile-time based on the passed argument types (not actual values).

Overload Resolution

A process of **selecting the most appropriate overload** at compile-time based on the passed argument types (not actual values).

The input for a process is a candidate set of function overloads found by the Name Lookup step.

Back To Basics: Overload Resolution - CppCon 2021



CppCon 2021 - Back To Basics

Back To Basics: Overload Resolution - CppCon 2021

Overload Resolution Process

Overload Resolution Process

- Removes invalid and not viable candidates from the set when

Overload Resolution Process

- **Removes invalid and not viable candidates** from the set when
  - candidates have *more parameters* than arguments being passed

Overload Resolution Process

- **Removes invalid and not viable candidates** from the set when
  - candidates have *more parameters* than arguments being passed
  - candidates have *less parameters*
    - unless default arguments exist in the function declaration

Overload Resolution Process

- **Removes invalid and not viable candidates** from the set when
  - candidates have *more parameters* than arguments being passed
  - candidates have *less parameters*
    - unless default arguments exist in the function declaration
  - candidate's associated *constraints are not satisfied*

Overload Resolution Process

- **Removes invalid and not viable candidates** from the set when
  - candidates have *more parameters* than arguments being passed
  - candidates have *less parameters*
    - unless default arguments exist in the function declaration
  - candidate's associated *constraints are not satisfied*
  - passed *arguments are not implicitly convertible* to function parameters' types

Overload Resolution Process

- **Removes invalid and not viable candidates** from the set when
  - candidates have *more parameters* than arguments being passed
  - candidates have *less parameters*
    - unless default arguments exist in the function declaration
  - candidate's associated *constraints are not satisfied*
  - passed *arguments are not implicitly convertible* to function parameters' types
- **Ranks** the remaining candidates via pair-wise comparisons

Overload Resolution Process

- **Removes invalid and not viable candidates** from the set when
  - candidates have *more parameters* than arguments being passed
  - candidates have *less parameters*
    - unless default arguments exist in the function declaration
  - candidate's associated *constraints are not satisfied*
  - passed *arguments are not implicitly convertible* to function parameters' types
- **Ranks** the remaining candidates via pair-wise comparisons
- If **exactly one** viable function **is better than all others**, this **function is called**
  - otherwise, **compilation fails**

Ranking

For each pair of viable function **F1** and **F2**, the **implicit conversion sequences from the i-th argument to i-th parameter are ranked** to determine which one is better.

Conversion ranks

RANK	CATEGORY	CONVERSIONS
1	exact match	no conversion required trivial conversions
2	promotions	integral or floating-point promotion integral or floating-point promotion + trivial conversions
3	conversions	standard conversion standard conversion + trivial conversions
4	user-defined	user-defined conversion user-defined conversion + trivial conversions user-defined conversion + standard conversion
5	ellipsis	... C-style function argument

Tie Breakers

If both candidates result in the same rank a tie breaker may run to find the winner.

Tie Breakers

If both candidates result in the same rank a tie breaker may run to find the winner.

- Less conversion steps wins
- Binding "rvalue reference to an rvalue" wins with binding "lvalue reference to an rvalue"
- Less cv-qualified reference or pointer wins
- ...

Tie Breakers

If both candidates result in the same rank a tie breaker may run to find the winner.

- Less conversion steps wins
- Binding "rvalue reference to an rvalue" wins with binding "lvalue reference to an rvalue"
- Less cv-qualified reference or pointer wins
- ...

If no tie breakers can be applied in such a situation, the call is ambiguous.

Examples: Standard Conversions

```
void foo(const int&);  // #1
void foo(const int&&); // #2

int i = 123;
foo(i);    // 'lvalue int -> const int&' is the only valid conversion
foo(123); // 'rvalue int -> const int&&' better than 'rvalue int -> const int&'
```

Examples: Standard Conversions

```cpp
void foo(const int&);  // #1
void foo(const int&&); // #2

int i = 123;
foo(i);   // 'lvalue int -> const int&' is the only valid conversion
foo(123); // 'rvalue int -> const int&&' better than 'rvalue int -> const int&'
```

```cpp
void foo(const int&); // #1
void foo(int);        // #2

int i = 123;
foo(i);   // error: call of overloaded 'foo(int&)' is ambiguous (rank Exact Match)
foo(123); // error: call of overloaded 'foo(int)' is ambiguous (rank Exact Match)
```

Examples: Standard Conversions

```
void foo(const int&);  // #1
void foo(const int&&); // #2

int i = 123;
foo(i);   // 'lvalue int -> const int&' is the only valid conversion
foo(123); // 'rvalue int -> const int&&' better than 'rvalue int -> const int&'
```

```
void foo(const int&); // #1
void foo(int);        // #2

int i = 123;
foo(i);   // error: call of overloaded 'foo(int&)' is ambiguous (rank Exact Match)
foo(123); // error: call of overloaded 'foo(int)' is ambiguous (rank Exact Match)
```

```
int f(const int*); // '&i -> const int*'
int f(int*);       // '&i -> int*'

int i;
int j = f(&i); //  calls f(int*)
```

Example: User-defined Conversions

```cpp
struct A {
  operator short();
};

int f(int);   // 'A -> short', followed by 'short -> int' (rank Promotion)
int f(float); // 'A -> short', followed by 'short -> float' (rank Conversion)

A a;
int i = f(a); // calls f(int)
```

Functions vs function templates

If a function and template function overloads are deemed equal, a *non-template function will be selected*.

Functions vs function templates

If a function and template function overloads are deemed equal, a *non-template function will be selected*.

```
template<typename T>
T from_string(std::string_view str);

struct converter; {
  std::string_view str;

  template<typename To>
  operator To() const { return from_string<To>(str); }
};

converter from_string(std::string_view str) { return converter{str}; }
```

Function templates vs function templates

If both candidates are template specializations, a *more specialized one* according to the partial ordering rules for template specializations *wins*.

Constrained no-template functions

If both are non-template functions with the same parameter-type-lists, a *more constrained one* according to the partial ordering of constraints *wins*.

Constrained no-template functions

```
template<auto V>
struct X {
  using type = decltype(V);

  static void foo()
  { std::cout << "Other\n"; }

  static void foo()
    requires std::integral<type>
  { std::cout << "Integral\n"; }

  static void foo()
    requires std::signed_integral<type>
  { std::cout << "Signed Integral\n"; }

  static void foo()
    requires std::floating_point<type>
  { std::cout << "Floating-point\n"; }
};
```

```
X<123>::foo();          // Signed Integral
X<123U>::foo();         // Integral
X<3.14>::foo();         // Floating-point
X<std::kilo{}>::foo(); // Other
```

Ranking for multiple arguments

In order for F1 to be determined a better function than F2 the implicit *conversions for all arguments of F1 must be not worse* than the corresponding conversions of F2 arguments and *at least conversion for one argument of F1 must be considered better*.

Ranking for multiple arguments

In order for F1 to be determined a better function than F2 the implicit *conversions for all arguments of F1 must be not worse* than the corresponding conversions of F2 arguments and *at least conversion for one argument of F1 must be considered better*.

```
void foo(int, double, int);     // #1
void foo(double, int, double); // #2
```

```
foo(42, 3.14, 42U);       // (R1, R1, R1) vs (R3, R3, R3) -> calls #1
foo(3.14, 42U, 12);       // (R3, R3, R1) vs (R1, R3, R3) -> ambiguous
foo(42, short{14}, 42); // (R1, R3, R1) vs (R3, R2, R3) -> ambiguous
```

Recommendation: Resolving ambiguity

Recommendation: Resolving ambiguity

1. Use `explicit` constructors and conversion operators by default

Recommendation: Resolving ambiguity

1. Use `explicit` constructors and conversion operators by default
2. Use constraints on function templates and regular functions whenever possible

Recommendation: Resolving ambiguity

1. Use `explicit` constructors and conversion operators by default
2. Use constraints on function templates and regular functions whenever possible
3. Explicitly convert function argument to required type on a function call site

Recommendation: Overload Resolution

Keep your overloads simple and easy. Do not make your life's and computer's work harder ;-)

Bonus: Member Access Rules Check

Member access is checked on the best candidate only after the Overload Resolution is done. If it requires the inaccessible member access, the compilation fails.

Bonus: Member Access Rules Check

Member access is checked on the best candidate only after the Overload Resolution is done. If it requires the inaccessible member access, the compilation fails.

```
struct X {
  void foo(int) { std::cout << "int\n"; }
private:
  void foo(double) { std::cout << "double\n"; }
};
```

```
X x;
x.foo(3.14);  // error: 'void X::foo(double)' is private within this context
```

CAUTION
Programming
is addictive
(and too much fun)