Calling Functions: A Tutorial

Kyle J. Knoepfel

Programming Video Journal Club, Session 8

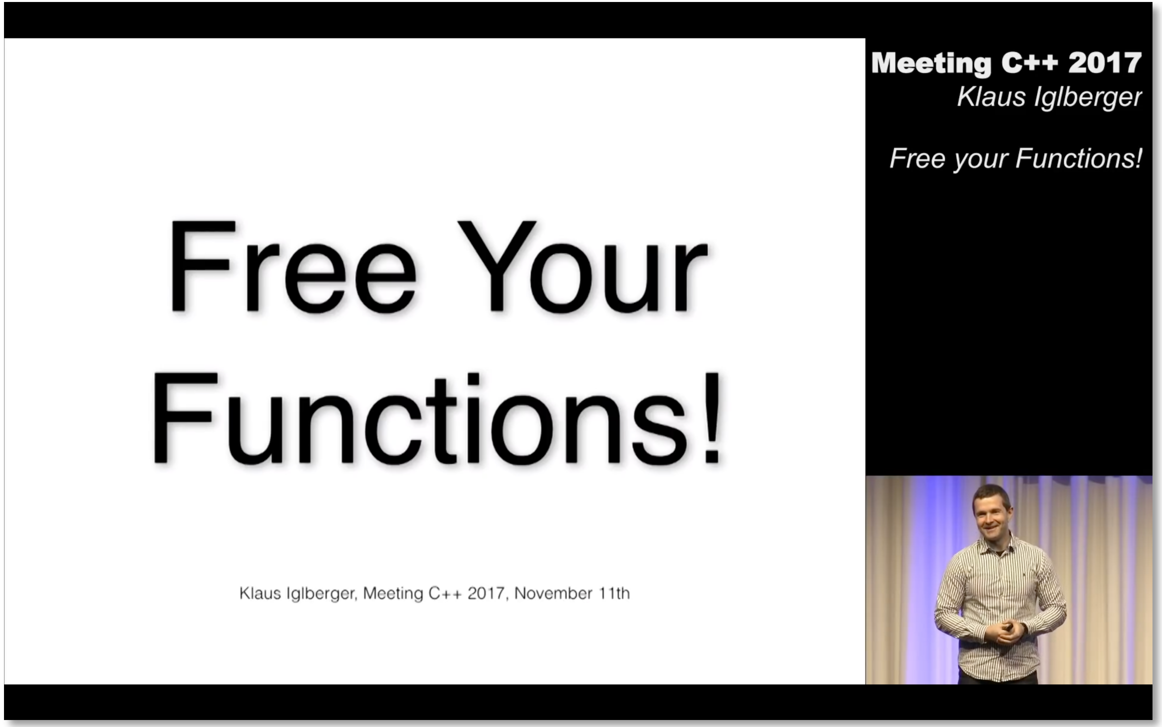
10 February 2021

Why this talk?

- We're used to calling functions all the time.
  - But do we understand how that works...or doesn't work?
- This talk showed me there were things I knew that "just ain't so."

Klaus Iglberger

- Presented a nice talk on free functions in 2017:

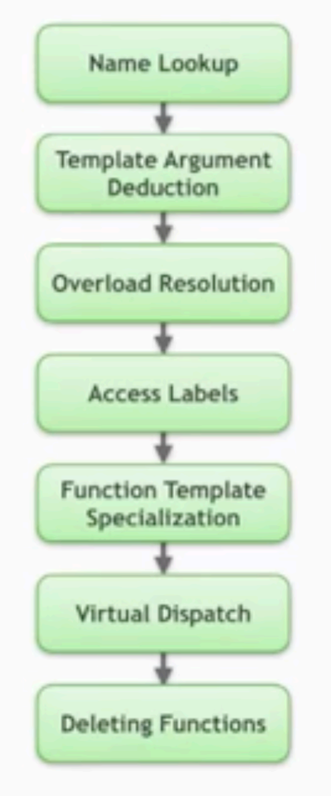


A prerequisite

- Talk assumes you know the types of literal values.

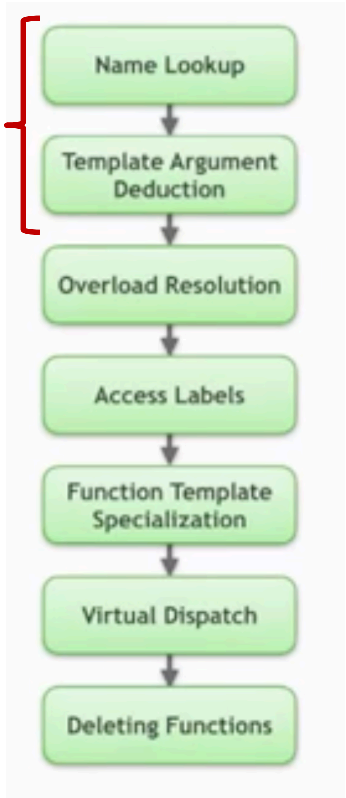
```
auto b = true;  // -> bool
auto i = 1;     // -> int
auto d = 2.0;   // -> double
auto s = "str"; // -> char const*
```

Klaus' Outline



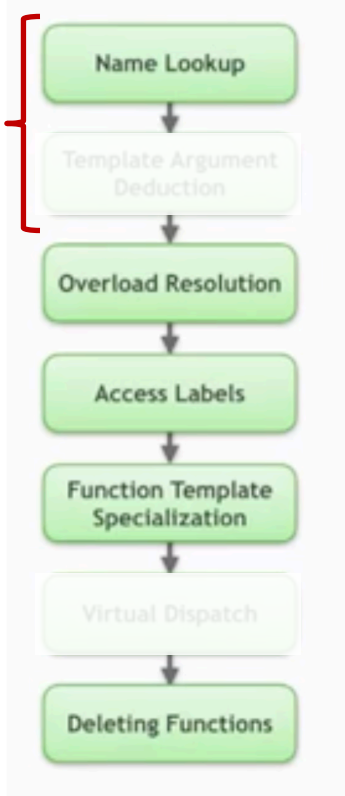
Klaus' Outline

Overload set formation



Klaus' Outline

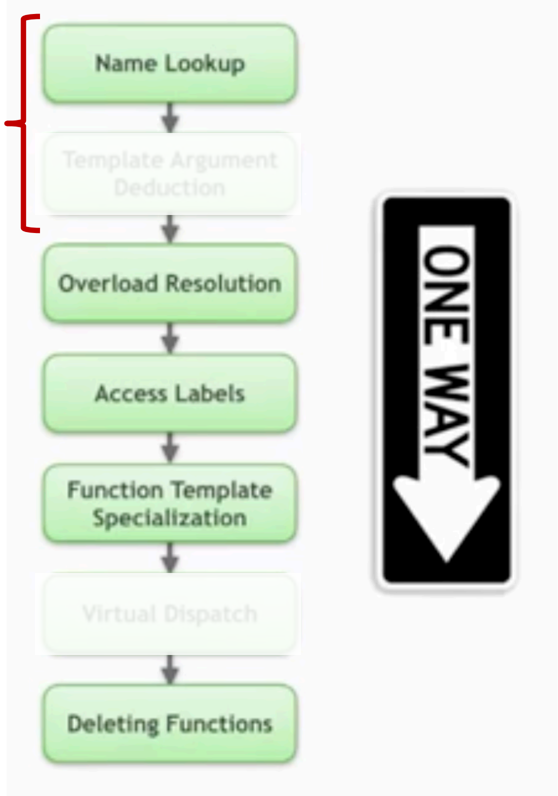
Overload set formation



Did not cover all steps due to time constraints.

Klaus' Outline

Overload set formation



Take-home point:

Once you've made it to one step, there's no going back.

Name lookup

Name lookup

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

- Applies to functions, variables, types, etc.

Name lookup

Name lookup

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

- Applies to functions, variables, types, etc.
- Are the following legal, and if so, what is the expected behavior?

Name lookup

Name lookup

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

- Applies to functions, variables, types, etc.
- Are the following legal, and if so, what is the expected behavior?

```
void f(int);
void g(int a)
{
  {
    int a = 42;
    f(a);
  }
}
```


Fermilab

Name lookup

Name lookup

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

- Applies to functions, variables, types, etc.
- Are the following legal, and if so, what is the expected behavior?

```
void f(int);
void g(int a)
{
   {
      int a = 42;
      f(a);
   }
}
```

Legal
Unused variable

Fermilab

Name lookup

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

- Applies to functions, variables, types, etc.
- Are the following legal, and if so, what is the expected behavior?

```
void f(int);
void g(int a)
{
  {
    int a = 42;
    f(a);
  }
}
```

Legal

Unused variable

```
int x;
int main()
{
  int x = x;
  return x;
}
```

Name lookup

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

- Applies to functions, variables, types, etc.
- Are the following legal, and if so, what is the expected behavior?

```
void f(int);
void g(int a)
{
  {
    int a = 42;
    f(a);
  }
}
```

Legal

Unused variable

```
int x;
int main()
{
  int x = x;
  return x;
}
```

Legal, but returns uninitialized value.

Name lookup

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

- Applies to functions, variables, types, etc.
- Are the following legal, and if so, what is the expected behavior?

```
void f(int);
void g(int a)
{
  {
    int a = 42;
    f(a);
  }
}
```

Legal

Unused variable

```
int x;
int main()
{
  int x = x;
  return x;
}
```

Legal, but returns uninitialized value.

```
class A {
public:
  A(int x) : x{x} {}
private:
  int x;
};
```

Name lookup

Name lookup

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

- Applies to functions, variables, types, etc.
- Are the following legal, and if so, what is the expected behavior?

```
void f(int);
void g(int a)
{
  {
    int a = 42;
    f(a);
  }
}
```

Legal

Unused variable

```
int x;
int main()
{
  int x = x;
  return x;
}
```

Legal, but returns uninitialized value.

```
class A {
public:
  A(int x) : x{x} {}
private:
  int x;
};
```

Legal, but confusing

Forming the overload set

```
#include <iostream>

#define PRINT_FUNCTION \
  std::cout << __PRETTY_FUNCTION__ << '\n'

namespace N {
  void f(double) { PRINT_FUNCTION; }
  namespace M {
    void f(int) { PRINT_FUNCTION; }
    void f(char const*) { PRINT_FUNCTION; }

    void g()
    {
      f(2.);
    }
  }
}

int main()
{
  N::M::g();
}
```

Forming the overload set

```
#include <iostream>

#define PRINT_FUNCTION \
  std::cout << __PRETTY_FUNCTION__ << '\n'

namespace N {
  void f(double) { PRINT_FUNCTION; }
  namespace M {
    void f(int) { PRINT_FUNCTION; }
    void f(char const*) { PRINT_FUNCTION; }

    void g()
    {
      f(2.);
    }
  }
}

int main()
{
  N::M::g();
}
```

```
void N::M::f(int)
```


Fermilab

Forming the overload set

```
#include <iostream>

#define PRINT_FUNCTION \
  std::cout << __PRETTY_FUNCTION__ << '\n'

namespace N {
  void f(double) { PRINT_FUNCTION; }
  namespace M {
    void f(int) { PRINT_FUNCTION; }
    void f(char const*) { PRINT_FUNCTION; }

    void g()
    {
      f(2.);
    }
  }
}

int main()
{
  N::M::g();
}
```

```
void N::M::f(int)
```

The overload set determined by searching scopes from “in-to-out.” The set at the call site is:

```
void N::M::f(int)
void N::M::f(char const*)
```


Fermilab

Forming the overload set

```
#include <iostream>

#define PRINT_FUNCTION \
  std::cout << __PRETTY_FUNCTION__ << '\n'

namespace N {
  void f(double) { PRINT_FUNCTION; }
  namespace M {
    void f(int) { PRINT_FUNCTION; }
    void f(char const*) { PRINT_FUNCTION; }

    void g()
    {
      f(2.);
    }
  }
}

int main()
{
  N::M::g();
}
```

Forming the overload set

```
#include <iostream>

#define PRINT_FUNCTION \
  std::cout << __PRETTY_FUNCTION__ << '\n'

namespace N {
  void f(double) { PRINT_FUNCTION; }
  namespace M {
    void f(int) { PRINT_FUNCTION; }
    void f(char const*) { PRINT_FUNCTION; }

    void g()
    {
      f(2.);
    }
  }
}

int main()
{
  N::M::g();
}
```

```
error: no matching function for call to 'f'
    f(2.);
    ^
note: candidate function not viable: no known
conversion from 'double' to 'const char *' for
1st argument
    void f(char const*) { PRINT_FUNCTION; }
```

Forming the overload set

```
#include <iostream>

#define PRINT_FUNCTION \
  std::cout << __PRETTY_FUNCTION__ << '\n'

namespace N {
  void f(double) { PRINT_FUNCTION; }
  namespace M {
    void f(int) { PRINT_FUNCTION; }
    void f(char const*) { PRINT_FUNCTION; }

    void g()
    {
      f(2.);
    }
  }
}

int main()
{
  N::M::g();
}
```

```
error: no matching function for call to 'f'
    f(2.);
    ^
note: candidate function not viable: no known
conversion from 'double' to 'const char *' for
1st argument
    void f(char const*) { PRINT_FUNCTION; }
```

The overload set determined by searching scopes from “in-to-out.” The set at the call site is:

`void N::M::f(char const*)`

No conversion from `double` to `char const*`.

Fermilab

Forming the overload set

```
#include <iostream>

#define PRINT_FUNCTION \
  std::cout << __PRETTY_FUNCTION__ << '\n'

namespace N {
  void f(double) { PRINT_FUNCTION; }
  namespace M {
    void f(int) { PRINT_FUNCTION; }
    void f(char const*) { PRINT_FUNCTION; }

    void g()
    {
      f(2.);
    }
  }
}

int main()
{
  N::M::g();
}
```

```
error: no matching function for call to 'f'
    f(2.);
    ^
note: candidate function not viable: no known
conversion from 'double' to 'const char *' for
1st argument
    void f(char const*) { PRINT_FUNCTION; }
```

The overload set determined by searching scopes from "in-to-out." The set at the call site is:

`void N::M::f(char const*)`

No conversion from `double` to `char const*`.

How do we call `N::f`?

Forming the overload set

```
#include <iostream>

#define PRINT_FUNCTION \
  std::cout << __PRETTY_FUNCTION__ << '\n'

namespace N {
  void f(double) { PRINT_FUNCTION; }
  namespace M {
    void f(int) { PRINT_FUNCTION; }
    void f(char const*) { PRINT_FUNCTION; }

    void g()
    {
      N::f(2.);
    }
  }
}

int main()
{
  N::M::g();
}
```

Qualified lookup. The overload set at the call site is:

```
void N::f(double)
```

Forming the overload set

```cpp
#include <iostream>

#define PRINT_FUNCTION \
  std::cout << __PRETTY_FUNCTION__ << '\n'

namespace N {
  void f(double) { PRINT_FUNCTION; }
  namespace M {
    void f(int) { PRINT_FUNCTION; }
    void f(char const*) { PRINT_FUNCTION; }

    void g()
    {
      using N::f; // using directive
      f(2.);
    }
  }
}

int main()
{
  N::M::g();
}
```

Unqualified lookup. The overload set at the call site is:

```
void N::f(double)
void N::M::f(int)
void N::M::f(char const*)
```

ADL *or* How do you check if `std::vector<int>` is empty?

```cpp
// 1. Explicit size check
if (v.size() == 0) { ... }

// 2. Let the container do the work
if (v.empty()) { ... }

// 3. Use the std::empty function template
if (std::empty(v)) { ... }

// 4. Same as 3 but rely on ADL
if (empty(v)) { ... }
```

ADL *or* How do you check if `std::vector<int>` is empty?

```cpp
// 1. Explicit size check
if (v.size() == 0) { ... }

// 2. Let the container do the work
if (v.empty()) { ... }

// 3. Use the std::empty function template
if (std::empty(v)) { ... }

// 4. Same as 3 but rely on ADL
if (empty(v)) { ... }
```

- At the very least, use the function whose name connotes the operation you want.
- Free-function versions allow for customization.
- Unqualified lookup allows for **argument-dependent lookup** (ADL) to take effect.

Do you have a preference?

Conversions

- Once the overload set has been formed, it resolve to one function implementation.
- This may involve conversions of the function arguments.

Conversions

- Once the overload set has been formed, it resolve to one function implementation.
- This may involve conversions of the function arguments.

Finding a Best Match (1 Parameter)

For a single argument, the compiler chooses the best available option:

1. **Exact/identity match**
2. **Trivial conversion**

Rank 1

3. **Promotion**
4. **Promotion + trivial conversion**

Rank 2

5. **Standard conversion**
6. **Standard conversion + trivial conversion**

Rank 3

7. **User-defined conversion**
8. **User-defined conversion + trivial conversion**
9. **User-defined conversion + standard conversion**

Fermilab

Conversions

- Once the overload set has been formed, it resolve to one function implementation.
- This may involve conversions of the function arguments.

Finding a Best Match (1 Parameter)

For a single argument, the compiler chooses the best available option:

1. **Exact/identity match**
2. **Trivial conversion**

Rank 1

3. **Promotion**
4. **Promotion + trivial conversion**

Rank 2

5. **Standard conversion**
6. **Standard conversion + trivial conversion**

Rank 3

7. **User-defined conversion**
8. **User-defined conversion + trivial conversion**
9. **User-defined conversion + standard conversion**

```
void f(A const&);
A a;
f(a);
```

Conversions

- Once the overload set has been formed, it resolve to one function implementation.
- This may involve conversions of the function arguments.

Finding a Best Match (1 Parameter)

For a single argument, the compiler chooses the best available option:

1. Exact/identity match
2. Trivial conversion

Rank 1

3. Promotion
4. Promotion + trivial conversion

Rank 2

```
signed short -> int
float -> double
```

5. Standard conversion
6. Standard conversion + trivial conversion

Rank 3

7. User-defined conversion
8. User-defined conversion + trivial conversion
9. User-defined conversion + standard conversion

Conversions

- Once the overload set has been formed, it resolve to one function implementation.
- This may involve conversions of the function arguments.

Finding a Best Match (1 Parameter)

For a single argument, the compiler chooses the best available option:

1. **Exact/identity match**
2. **Trivial conversion**

Rank 1

3. **Promotion**
4. **Promotion + trivial conversion**

Rank 2

5. **Standard conversion**
6. **Standard conversion + trivial conversion**

Rank 3

```
float -> int
decltype(nullptr) -> A*
```

7. **User-defined conversion**
8. **User-defined conversion + trivial conversion**
9. **User-defined conversion + standard conversion**

Conversions

- Once the overload set has been formed, it resolve to one function implementation.
- This may involve conversions of the function arguments.

Finding a Best Match (1 Parameter)

For a single argument, the compiler chooses the best available option:

1. **Exact/identity match**
2. **Trivial conversion**

Rank 1

3. **Promotion**
4. **Promotion + trivial conversion**

Rank 2

5. **Standard conversion**
6. **Standard conversion + trivial conversion**

Rank 3

7. **User-defined conversion**
8. **User-defined conversion + trivial conversion**
9. **User-defined conversion + standard conversion**

```
class A {
public:
  explicit operator bool() const;
};
```

Access labels

```
class Object
{
 public:
   void f( int );      // (1)
 private:
   void f( double );   // (2)
};


Object obj{};
obj.f( 1.0 ); // (2) is selected; access violation!
```


Fermilab

Access labels

```
class Object
{
 public:
   void f( int );      // (1)
 private:
   void f( double );   // (2)
};


Object obj{};
obj.f( 1.0 ); // (2) is selected; access violation!
```

"It's an access label, not a visibility label."

–K. Iglberger

Function template specialization

- Klaus did not spend much time on this.
- An instantiated template is chosen as part of the overload set early on.
- The specialization is selected late in the process.

```
template< typename T > void f( T );   // (1)

template< > void f( char* );          // (2)

template< typename T > void f( T* );  // (3)


int main()
{
   char* cp{ nullptr };

   f( cp );    // Calls function (3)
}
```

Function template specialization

- Klaus did not spend much time on this.
- An instantiated template is chosen as part of the overload set early on.
- The specialization is selected late in the process.

```
template< typename T > void f( T );    // (1)

template< > void f( char* );           // (2)

template< typename T > void f( T* );  // (3)


int main()
{
   char* cp{ nullptr };

   f( cp );    // Calls function (3)
}
```

- **Take-home point**: Avoid specializations if you can. *Is he right?*

Aside: Template selection with `nullptr`

```cpp
namespace {
  template <typename T> void f(T);  // (1)
  template <typename T> void f(T*); // (2)
}

int main()
{
  f(nullptr);  // is (1) or (2) instantiated/called?
}
```

Aside: Template selection with `nullptr`

```
namespace {
  template <typename T> void f(T);  // (1)
  template <typename T> void f(T*); // (2)
}

int main()
{
  f(nullptr);  // (1) is instantiated/called
}
```

Aside: Template selection with `nullptr`

```
namespace {
  template <typename T> void f(T);  // (1)
  template <typename T> void f(T*); // (2)
}

int main()
{
  f(nullptr);  // (1) is instantiated/called
}
```

- **Reason:** the type of `nullptr` is *not* a pointer. It is of type `std::nullptr_t`.

Deleting functions

- Like private member functions, deleted functions are still visible.

Deleting functions

- Like private member functions, deleted functions are still visible.

```
class A {
public:
  explicit A(std::vector<int> nums) :
    nums_{move(nums)}
  {}

  // Disable copying
  A(A const&) = delete;
  A& operator=(A const&) = delete;

private:
  std::vector<int> nums_;
};

A a({1,3});
auto b = a;
```

Deleting functions

- Like private member functions, deleted functions are still visible.

```
class A {
public:
  explicit A(std::vector<int> nums) :
    nums_{move(nums)}
  {}

  // Disable copying
  A(A const&) = delete;
  A& operator=(A const&) = delete;

private:
  std::vector<int> nums_;
};

A a({1,3});
auto b = a;
```

```
error: use of deleted function 'A::A(const A&)'
22 | auto b = a;
   |          ^
<source>:8:3: note: declared here
 8 | A(A const&) = delete;
   | ^
```

Deleting functions

- Like private member functions, deleted functions are still visible.

```cpp
class A {
public:
  explicit A(std::vector<int> nums) :
    nums_{move(nums)}
  {}

  // Disable copying
  A(A const&) = delete;
  A& operator=(A const&) = delete;

private:
  std::vector<int> nums_;
};

A a({1,3});
auto b = std::move(a);
```

Deleting functions

- Like private member functions, deleted functions are still visible.

```cpp
class A {
public:
  explicit A(std::vector<int> nums) :
    nums_{move(nums)}
  {}

  // Disable copying
  A(A const&) = delete;
  A& operator=(A const&) = delete;

private:
  std::vector<int> nums_;
};

A a({1,3});
auto b = std::move(a);
```

```
error: use of deleted function 'A::A(const A&)'
22 | auto b = std::move(a);
   |                     ^
<source>:8:3: note: declared here
 8 | A(A const&) = delete;
   | ^
```


Fermilab

Deleting functions

- Like private member functions, deleted functions are still visible.

```cpp
class A {
public:
  explicit A(std::vector<int> nums) :
    nums_{move(nums)}
  {}

  // Disable copying
  A(A const&) = delete;
  A& operator=(A const&) = delete;

private:
  std::vector<int> nums_;
};

A a({1,3});
auto b = std::move(a);
```

```
error: use of deleted function 'A::A(const A&)'
22 | auto b = std::move(a);
   |                     ^
<source>:8:3: note: declared here
 8 | A(A const&) = delete;
   | ^
```

What is the overload set here?

Fermilab

Deleting functions

- Like private member functions, deleted functions are still visible.

Implicitly-declared move constructor

If no user-defined move constructors are provided for a class type (`struct` , `class` , or `union`), and all of the following is true:

- there are no user-declared copy constructors;
- there are no user-declared copy assignment operators;
- there are no user-declared move assignment operators;
- there is no user-declared destructor.

then the compiler will declare a move constructor as a non-explicit `inline public` member of its class with the signature `T::T(T&&)`.

```
};

A a({1,3});
auto b = std::move(a);
```

Deleting functions

- Like private member functions, deleted functions are still visible.

Implicitly-declared move constructor

If no user-defined move constructors are provided for a class type (`struct`, `class`, or `union`), and all of the following is true:

- there are no user-declared copy constructors;
- there are no user-declared copy assignment operators;
- there are no user-declared move assignment operators;
- there is no user-declared destructor.

then the compiler will declare a move constructor as a non-explicit `inline public` member of its class with the signature `T::T(T&&)`.

`A(A const&) = delete;` is a user-declared copy constructor.

```
};

A a({1,3});
auto b = std::move(a);
```


Fermilab

Deleting functions

- Like private member functions, deleted functions are still visible.

Implicitly-declared move constructor

If no user-defined move constructors are provided for a class type (`struct`, `class`, or `union`), and all of the following is true:

- there are no user-declared copy constructors;
- there are no user-declared copy assignment operators;
- there are no user-declared move assignment operators;
- there is no user-declared destructor.

then the compiler will declare a move constructor as a non-explicit `inline public` member of its class with the signature `T::T(T&&)`.

`A(A const&) = delete;` is a user-declared copy constructor.

```
};

A a({1,3});
auto b = std::move(a);
```

A has no move constructor; `A(A&&)` is not even deleted.

Deleting functions

- Like private member functions, deleted functions are still visible.

```
class A {
public:
  explicit A(std::vector<int> nums) :
    nums_{move(nums)}
  {}

  // Disable copying
  A(A const&) = delete;
  A& operator=(A const&) = delete;

private:
  std::vector<int> nums_;
};

A a({1,3});
auto b = std::move(a);
```

```
error: use of deleted function 'A::A(const A&)'
22 | auto b = std::move(a);
   |                     ^
<source>:8:3: note: declared here
 8 | A(A const&) = delete;
   | ^
```

What is the overload set here?

Fermilab

Deleting functions

- Like private member functions, deleted functions are still visible.

```
class A {
public:
  explicit A(std::vector<int> nums) :
    nums_{move(nums)}
  {}

  // Disable copying and moving
  A(A const&) = delete;
  A& operator=(A const&) = delete;

private:
  std::vector<int> nums_;
};

A a({1,3});
auto b = std::move(a);
```

```
error: use of deleted function 'A::A(const A&)'
22 | auto b = std::move(a);
   |                     ^
<source>:8:3: note: declared here
 8 | A(A const&) = delete;
   | ^
```

What is the overload set here?

Fermilab

Deleting functions

- Like private member functions, deleted functions are still visible.

```
class A {
public:
  explicit A(std::vector<int> nums) :
    nums_{move(nums)}
  {}

  // Disable copying and moving
  A(A const&) = delete;
  A& operator=(A const&) = delete;

private:
  std::vector<int> nums_;
};

A a({1,3});
auto b = std::move(a);
```

```
error: use of deleted function 'A::A(const A&)'
22 | auto b = std::move(a);
   |                     ^
<source>:8:3: note: declared here
 8 | A(A const&) = delete;
   | ^
```

What is the overload set here?

`A(A const&)`

Fermilab

Deleting functions

- Like private member functions, deleted functions are still visible.

```
class A {
public:
  explicit A(std::vector<int> nums) :
    nums_{move(nums)}
  {}

  // Disable copying and moving
  A(A const&) = delete;
  A& operator=(A const&) = delete;

private:
  std::vector<int> nums_;
};

A a({1,3});
auto b = std::move(a);
```

```
error: use of deleted function 'A::A(const A&)'
22 | auto b = std::move(a);
   |                     ^
<source>:8:3: note: declared here
 8 | A(A const&) = delete;
   | ^
```

What is the overload set here?

`A(A const&)`

That's not what I wanted!

Deleting functions

- Like private member functions, deleted functions are still visible.

```
class A {
public:
  explicit A(std::vector<int> nums) :
    nums_{move(nums)}
  {}

  // Enable moving, but disable copying
  A(A&&) = default;
  A& operator=(A&&) = default;

private:
  std::vector<int> nums_;
};

A a({1,3});
auto b = std::move(a); // Ok
auto c = b; // Error, deleted copy c'tor
```

Deleting functions

- Like private member functions, deleted functions are still visible.

```
class A {
public:
  explicit A(std::vector<int> nums) :
    nums_{move(nums)}
  {}

  // Enable moving, but disable copying
  A(A&&) = default;
  A& operator=(A&&) = default;

private:
  std::vector<int> nums_;
};

A a({1,3});
auto b = std::move(a); // Ok
auto c = b; // Error, deleted copy c'tor
```

Take-home point:

Deleting functions does not remove them—it just makes it impossible to call them.

Some more take-home messages

- C++ has a well-defined procedure for selecting a function corresponding to a name.
  - It can be complicated!

- Many of these complications go away when meaningful and well-organized function names are chosen.

- I hope you had fun!