Calling Functions

A Tutorial

Klaus Iglberger, Meeting C++ 2020

klaus.iglberger@gmx.de

C++ Trainer/Consultant

Author of the blaze C++ math library

(Co-)Organizer of the Munich C++ user group

Regular presenter at C++ conferences

Email: klaus.iglberger@gmx.de

Klaus Iglberger

Content

- Overview
- Name Lookup
  - (Un-)Qualified Name Lookup
  - Argument Dependent Lookup
  - Two-Phase Lookup
- Template Argument Deduction
  - SFINAE
- Overload Resolution
  - (Viable) Candidate Functions
  - Ranking
  - Ambiguous Function Calls
- Access Labels
- Function Template Specializations
- Virtual Dispatch
- Deleting Functions

Content

- Overview
- Name Lookup
  - (Un-)Qualified Name Lookup
  - Argument Dependent Lookup
  - Two-Phase Lookup
- Template Argument Deduction
  - SFINAE
- Overload Resolution
  - (Viable) Candidate Functions
  - Ranking
  - Ambiguous Function Calls
- Access Labels
- Function Template Specializations
- Virtual Dispatch
- Deleting Functions

Disclaimer

- This talk does …
  - … focus on the basic mechanics of calling functions;
  - … point out the surprising details;
  - … give further references whenever necessary.
- This talk does **not** …
  - … show all possible examples;
  - … mention every single detail;
  - … try to be bleeding edge.

Overview

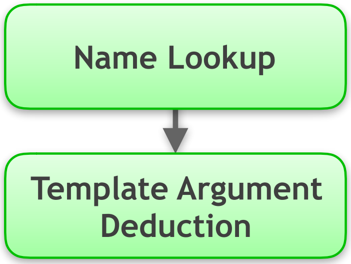
Overview

Name Lookup

Name Lookup

Select all (visible) candidate functions with a certain name within the current scope. If none is found, proceed into the next surrounding scope. This results in an overload set.

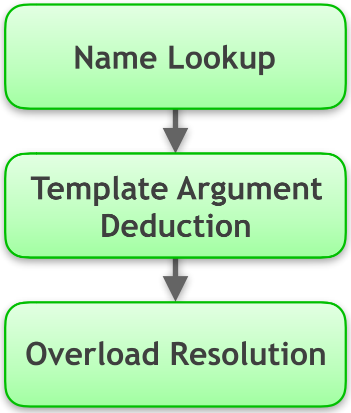
Overview



Template Argument Deduction

For all candidate function templates, deduce all function template parameters based on the given template arguments. and add them to the overload set. Substitution errors are not necessarily flagged as error (SFINAE).

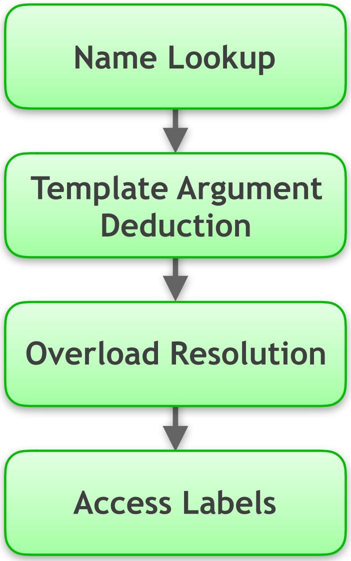
Overview



Overload Resolution

Find the best match from the overload set of candidate functions. If necessary, apply some necessary argument conversions.

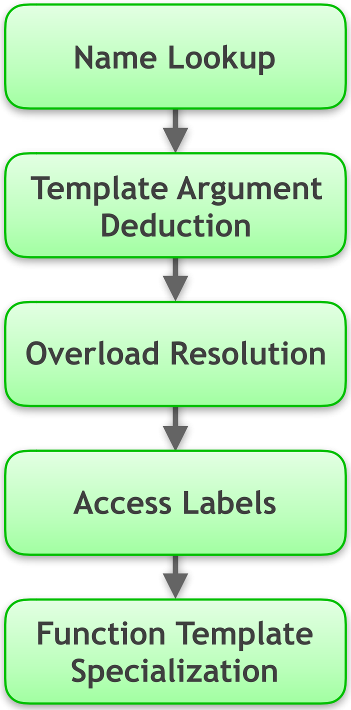
Overview



Access Labels

Check if the best match is accessible from the given call site.

Overview



Name Lookup

Template Argument Deduction

Overload Resolution

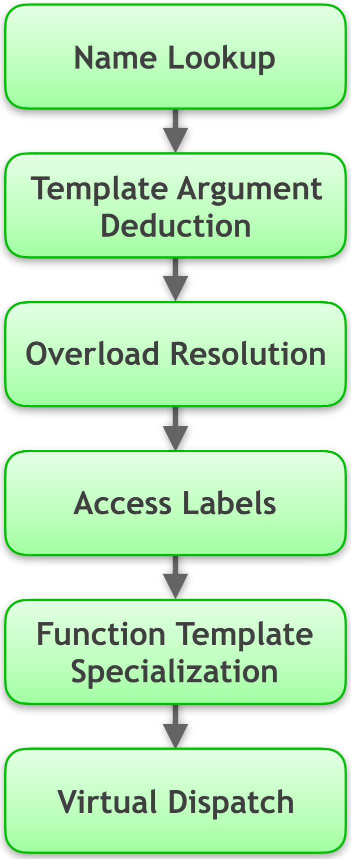
Access Labels

Function Template Specialization

Function Template Specialization

If the best match results from a template, choose the final function from the set of all specializations of the selected function template.

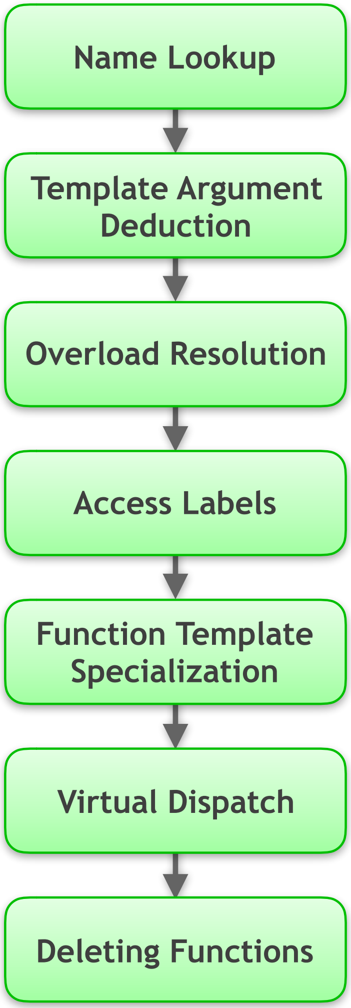
Overview



Virtual Dispatch

If the best match is a virtual function, select the most derived version of a virtual function.

Overview



Name Lookup

Template Argument Deduction

Overload Resolution

Access Labels

Function Template Specialization

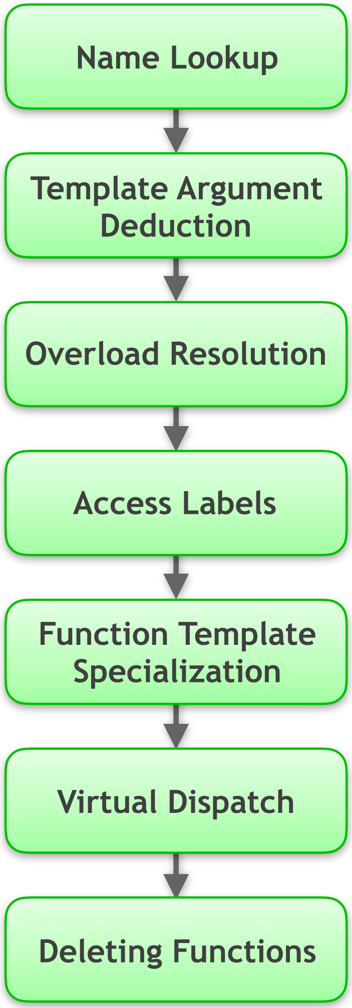
Virtual Dispatch

Deleting Functions

Deleting Functions

Check if the best match has been explicitly deleted (via =`delete`).

Overview



Name Lookup

Template Argument Deduction

Overload Resolution

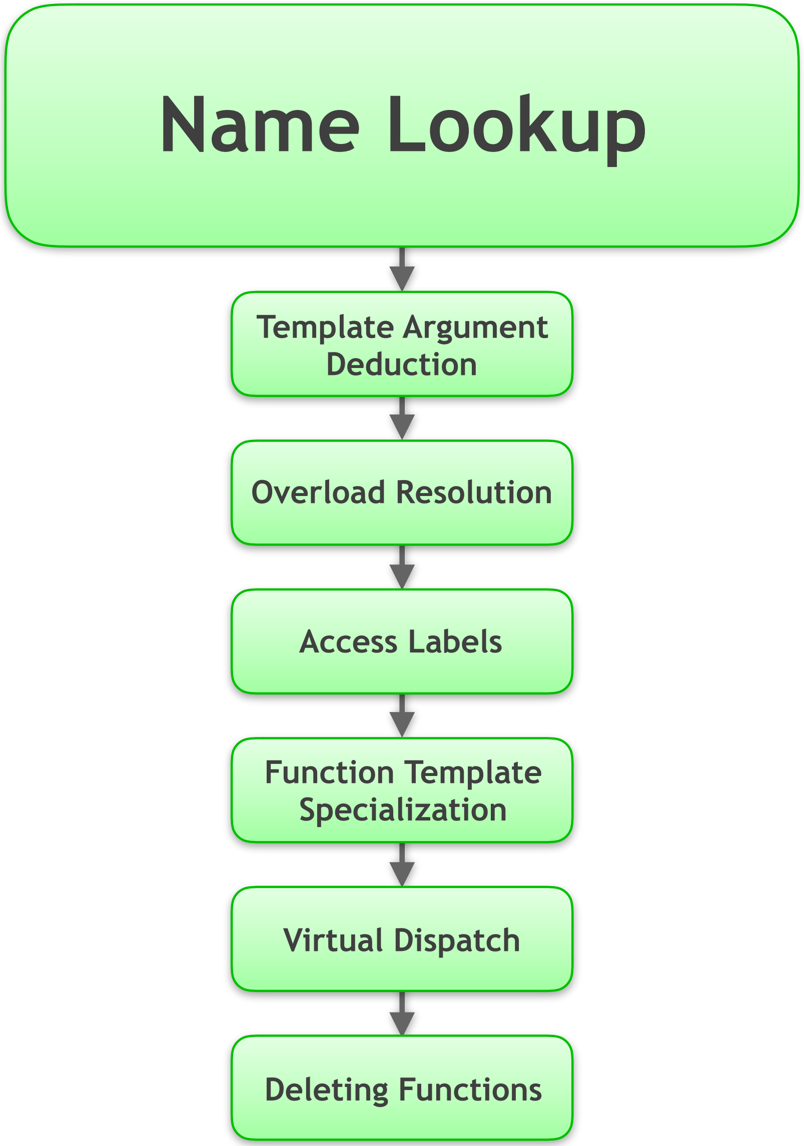
Access Labels

Function Template Specialization

Virtual Dispatch

Deleting Functions

Name Lookup



Name Lookup

Template Argument Deduction

Overload Resolution

Access Labels

Function Template Specialization

Virtual Dispatch

Deleting Functions

(Un-)Qualified Name Lookup

```
void f( double );     // (1)


namespace N1 {
   void f( int );     // (2)




} // namespace N1

int main()
{
   f( 1.0 );     // Unqualified lookup; calls (1)
   f( 42 );      // Unqualified lookup; calls (1)
   N1::f( 42 ); // Qualified lookup; calls (2)
}
```

(Un-)Qualified Name Lookup

```
void f( double );     // (1)



namespace N1 {







} // namespace N1


int main()
{
   f( 1.0 );     // Unqualified lookup; calls (1)
   f( 42 );      // Unqualified lookup; calls (1)
   N1::f( 42 );  // Ill-formed, no function found in ‘N1’
}
```

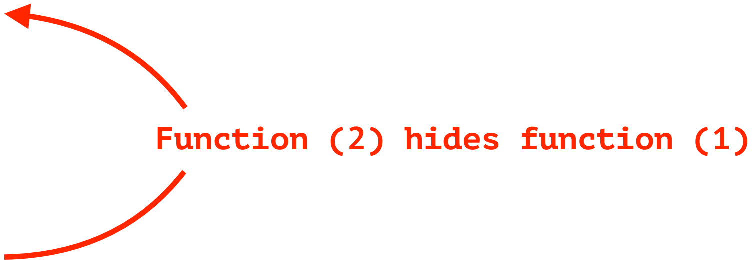
(Un-)Qualified Name Lookup

```
void f( double );      // (1)


namespace N1 {

   void f( int );      // (2)

   void g() { N1::f( 1.0 ); }
   void h() {     f( 1.0 ); }


} // namespace N1


int main()
{
   N1::g();      // Qualified lookup; calls (2)
   N1::h();      // Unqualified lookup; calls (2)

}
```

Function (2) hides function (1)

(Un-)Qualified Name Lookup

```
void f( double );     // (1)


namespace N1 {
   void f( string );  // (2)

   void g() { N1::f( 1.0 ); }
   void h() {     f( 1.0 ); }


} // namespace N1


int main()
{
   N1::g();      // Ill-formed, no conversion available
   N1::h();      // Ill-formed, no conversion available

}
```



Function (2) hides function (1)

(Un-)Qualified Name Lookup

```
void f( double );      // (1)


namespace N1 {

   constexpr int f=1; // (2)

   void g() { N1::f( 1.0 ); }
   void h() {     f( 1.0 ); }


} // namespace N1


int main()
{
   N1::g();      // Ill-formed, f cannot be used as a function
   N1::h();      // Ill-formed, f cannot be used as a function

}
```

Variable (2) hides function (1)

(Un-)Qualified Name Lookup

```
void f( double );     // (1)


namespace N1 {


   void g() { N1::f( 1.0 ); }
   void h() {     f( 1.0 ); }


} // namespace N1


int main()
{
   N1::g();      // Ill-formed, no function found in ‘N1'
   N1::h();      // Unqualified lookup; calls (1)

}
```

(Un-)Qualified Name Lookup

```
void f( double );     // (1)



namespace N1 {
   void f( int );     // (2)

   void g() {   ::f( 1.0 ); }
   void h() {     f( 1.0 ); }


} // namespace N1

int main()
{
   N1::g();      // Qualified lookup; calls (1)
   N1::h();      // Unqualified lookup; calls (2)

}
```

(Un-)Qualified Name Lookup

```
void f( double );     // (1)


namespace N1 {
   void f( int );     // (2)

   void g() {   ::f( 1.0 ); }
   void h() {     f( 1.0 ); }

   void f( double );  // (3)
} // namespace N1


int main()
{
   N1::g();      // Qualified lookup; calls (1)
   N1::h();      // Unqualified lookup; calls (2)

}
```

(Un-)Qualified Name Lookup

```
void f( double );     // (1)


namespace N1 {

   void f( int );     // (2)

   struct S {
      void f( int );  // (3)
      void g() { f( 1.0 ); }
   };

   void f( double );  // (4)

} // namespace N1


int main()
{
   N1::S s{};
   s.g();        // Unqualified lookup; calls (3)

}
```

Function (3) hides functions (1) and (2)

(Un-)Qualified Name Lookup

```
void f( double );     // (1)


namespace N1 {
   void f( int );     // (2)
   struct S {

      void g() { f( 1.0 ); }
   };
   void f( double );  // (4)
} // namespace N1


int main()
{
   N1::S s{};
   s.g();          // Unqualified lookup; calls (2)

}
```

Function (2) hides function (1)

(Un-)Qualified Name Lookup

```
void f( double );     // (1)



namespace N1 {


   struct S {

      void g() { f( 1.0 ); }
   };

   void f( double );  // (4)

} // namespace N1


int main()
{
   N1::S s{};
   s.g();        // Unqualified lookup; calls (1)

}
```

(Un-)Qualified Name Lookup

```
class Base
{
   // …
   virtual void f( int );      // (1)
   virtual void f( double );   // (2)
};


class Derived : public Base
{
   void f( double ) override;  // (3)
};



int main()
{
   Derived d{};

   d.f( 42 );    // Calls (3)
}
```

Function (3) hides function (1) and (2)

Argument Dependent Lookup

```
void f( double );     // (1)


namespace N1 {
   void f( int );     // (2)
   struct S {};
   void f( S );       // (3)

} // namespace N1


int main()
{
   N1::S s{};
   f( s );       // Argument dependent lookup (ADL); calls (3)

}
```

Argument Dependent Lookup

```
void f( double );     // (1)
template< typename T > void g( T t ) { f( t ); }

namespace N1 {
   void f( int );     // (2)
   struct S {};
   void f( S );       // (3)

} // namespace N1


int main()
{
   N1::S s{};
   g( s );       // Argument dependent lookup (ADL); calls (3)

}
```

Argument Dependent Lookup

```
void f( double );     // (1)
template< typename T > void g( T t ) { f( t ); }

namespace N1 {
   void f( int );     // (2)
   struct S {};
   void f( S );       // (3)

} // namespace N1
void f( N1::S );      // (4)

int main()
{
   N1::S s{};
   g( s );       // Ambiguous function call between (3) and (4)

}
```

Guidelines

Guideline: Remember that ADL only works for user-defined types.

Argument Dependent Lookup

```
namespace N1 {

   struct S {};

   void swap( S&, S& );  // (1)

} // namespace N1

template< typename T > void g( T& a, T& b )
{

   std::swap( a, b );
}


int main()
{
   N1::S s1{};
   N1::S s2{};

   g( s1, s2 );   // Qualified lookup, calls std::swap
}
```

Argument Dependent Lookup

```
namespace N1 {

   struct S {};
   void swap( S&, S& );  // (1)

} // namespace N1

template< typename T > void g( T& a, T& b )
{

   std::swap( a, b );
}


int main()
{
   N1::S s1{};
   N1::S s2{};

   g( s1, s2 );   // Qualified lookup, calls std::swap
}
```

Argument Dependent Lookup

```
namespace N1 {

   struct S {};

   void swap( S&, S& );  // (1)

} // namespace N1

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


int main()
{
   N1::S s1{};
   N1::S s2{};

   g( s1, s2 );   // Unqualified lookup, calls swap(S,S)
}
```

Guidelines

Guideline: Prefer unqualified name lookup to qualified name lookup.

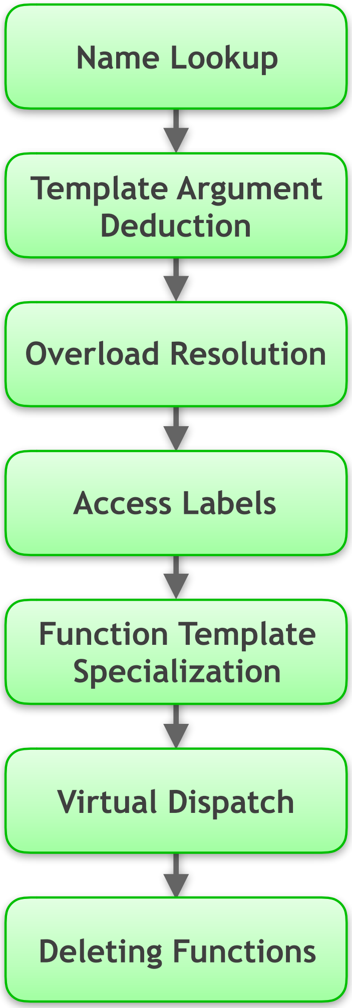
Core Guideline C.165: Use `using` for customisation points

Two-Phase Lookup

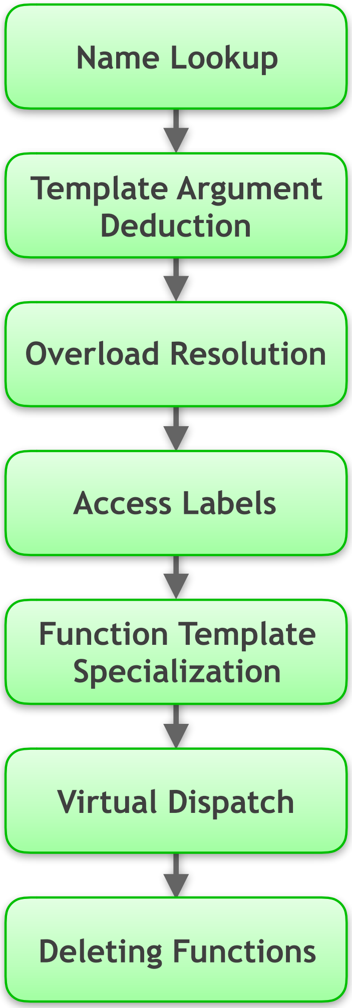
```cpp
void f( double );     // (1)
template< typename T > void g( T t ) { f( t ); }
void f( int );        // (2)
namespace N1 {

   struct S {};
   void f( S );       // (3)

} // namespace N1


int main()
{
   N1::S s{};
   g( s );       // Argument dependent lookup (ADL); calls (3)
   g( 42 );      // Regular lookup (no ADL); calls (1)
}
```

Questions?

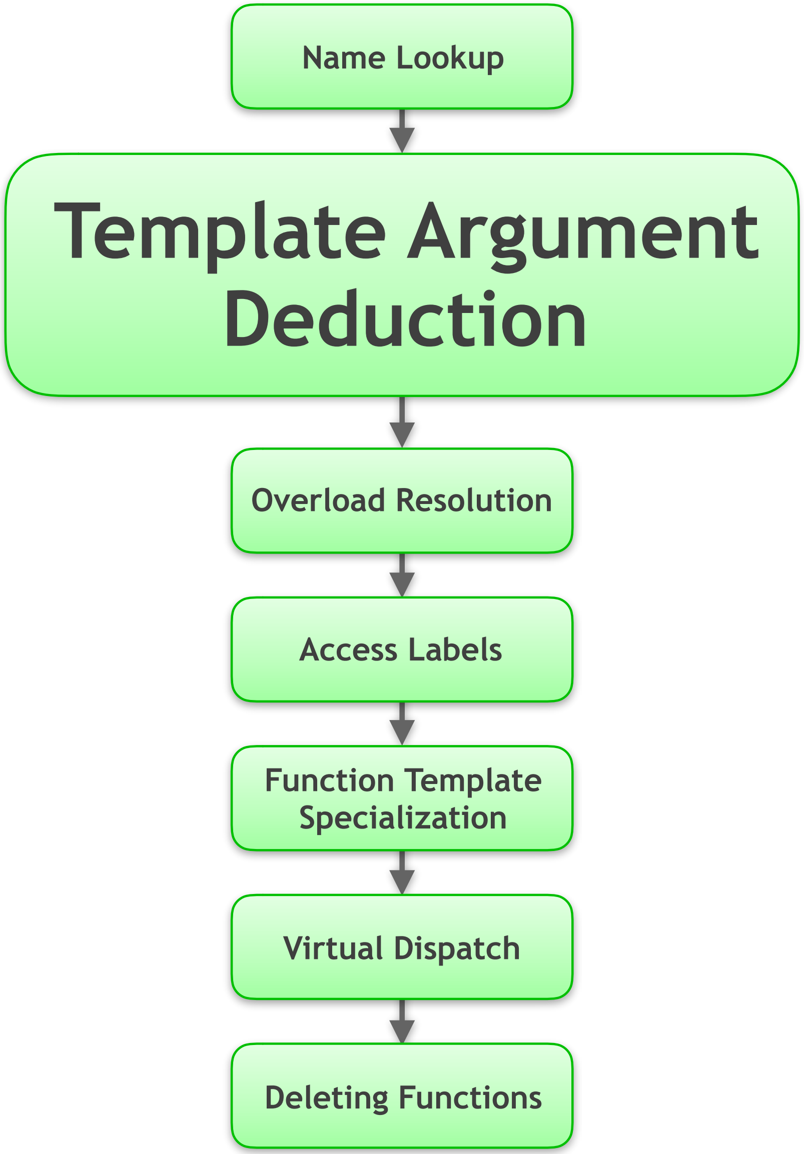
Name Lookup


Name Lookup
Template Argument Deduction
Overload Resolution
Access Labels
Function Template Specialization
Virtual Dispatch
Deleting Functions

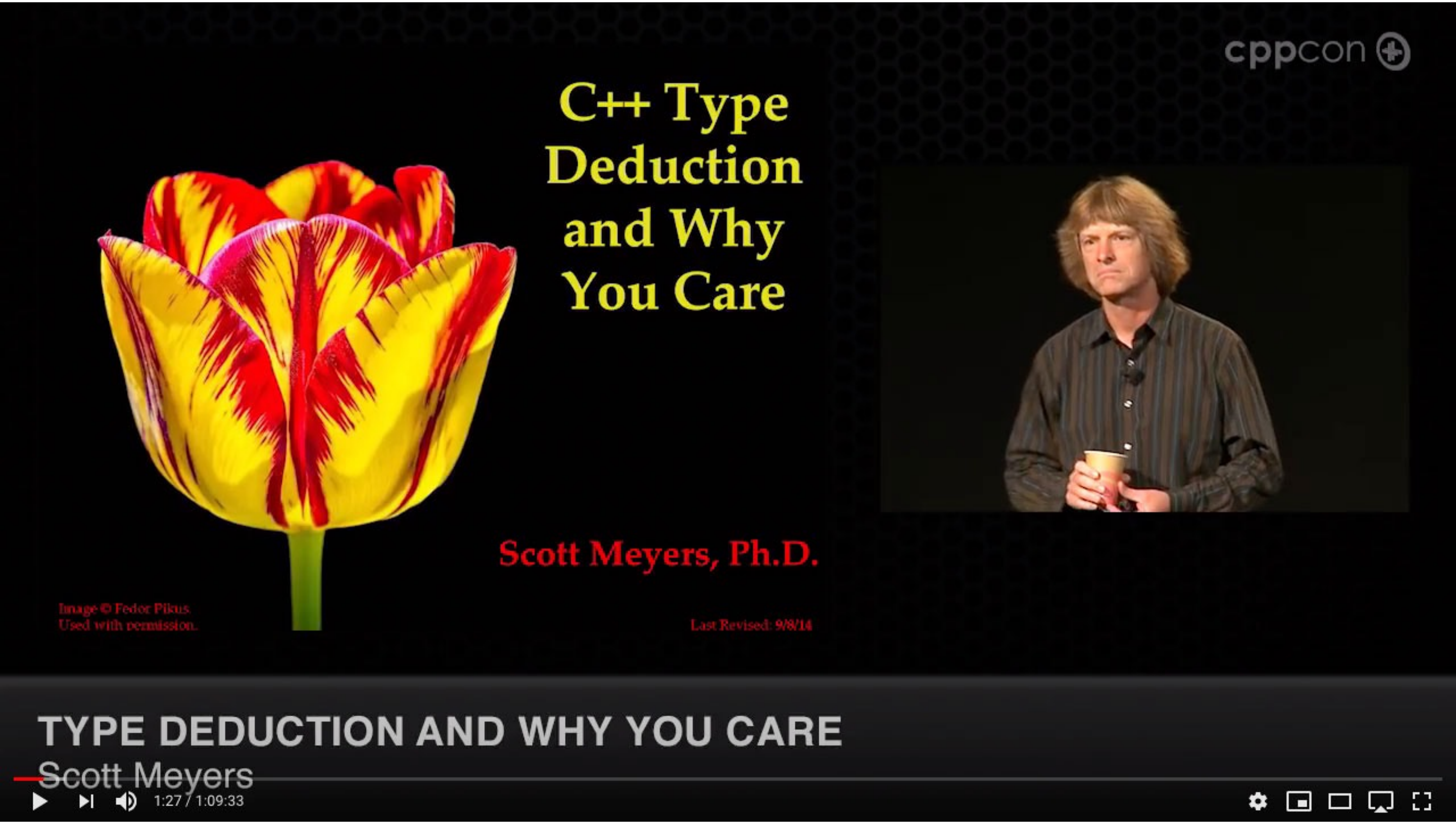
Name Lookup


Name Lookup
Template Argument Deduction
Overload Resolution
Access Labels
Function Template Specialization
Virtual Dispatch
Deleting Functions

Template Argument Deduction



Template Argument Deduction



C++ Type Deduction and Why You Care

Scott Meyers, Ph.D.

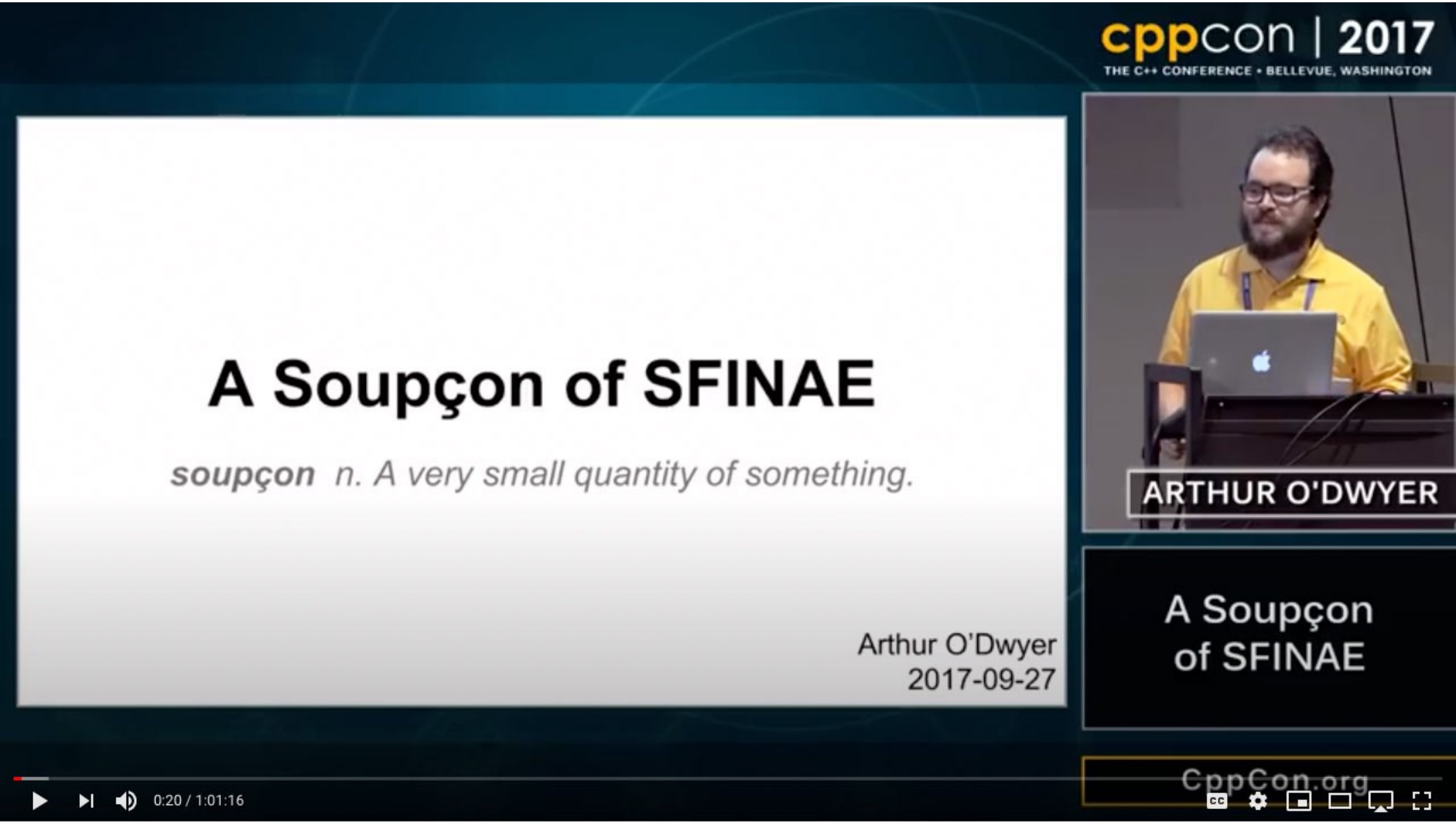
Image © Fedor Pikus.
Used with permission.

Last Revised: 9/8/14

TYPE DEDUCTION AND WHY YOU CARE

Scott Meyers

Template Argument Deduction

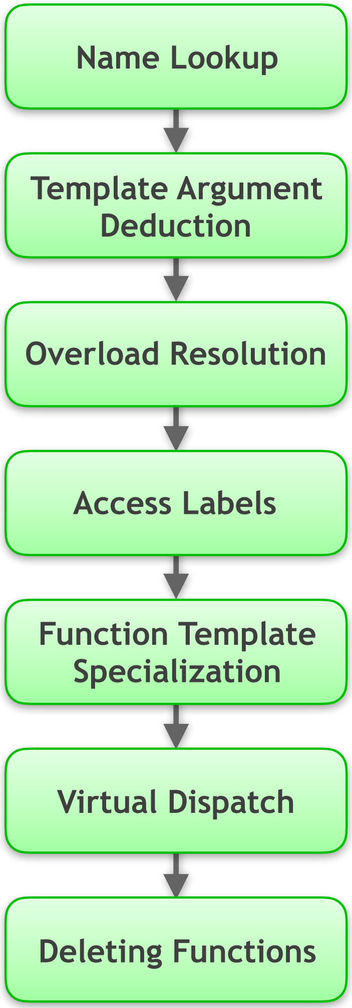


A Soupçon of SFINAE

soupçon *n. A very small quantity of something.*

Arthur O'Dwyer
2017-09-27

Template Argument Deduction



Name Lookup

Template Argument Deduction

Overload Resolution

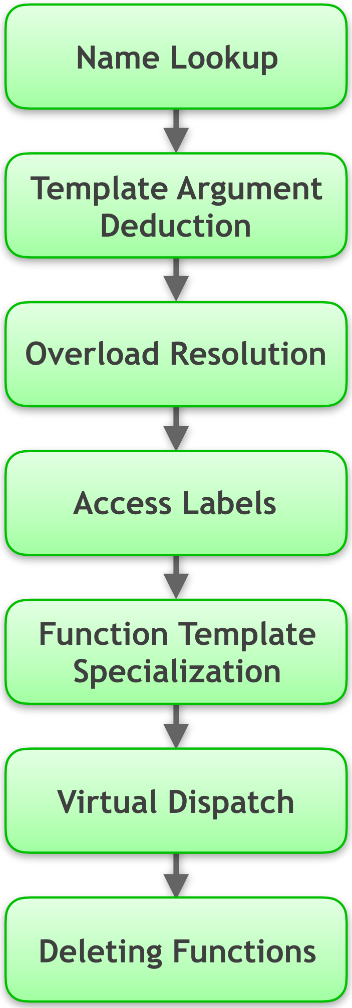
Access Labels

Function Template Specialization

Virtual Dispatch

Deleting Functions

Template Argument Deduction


Name Lookup

Template Argument Deduction

Overload Resolution

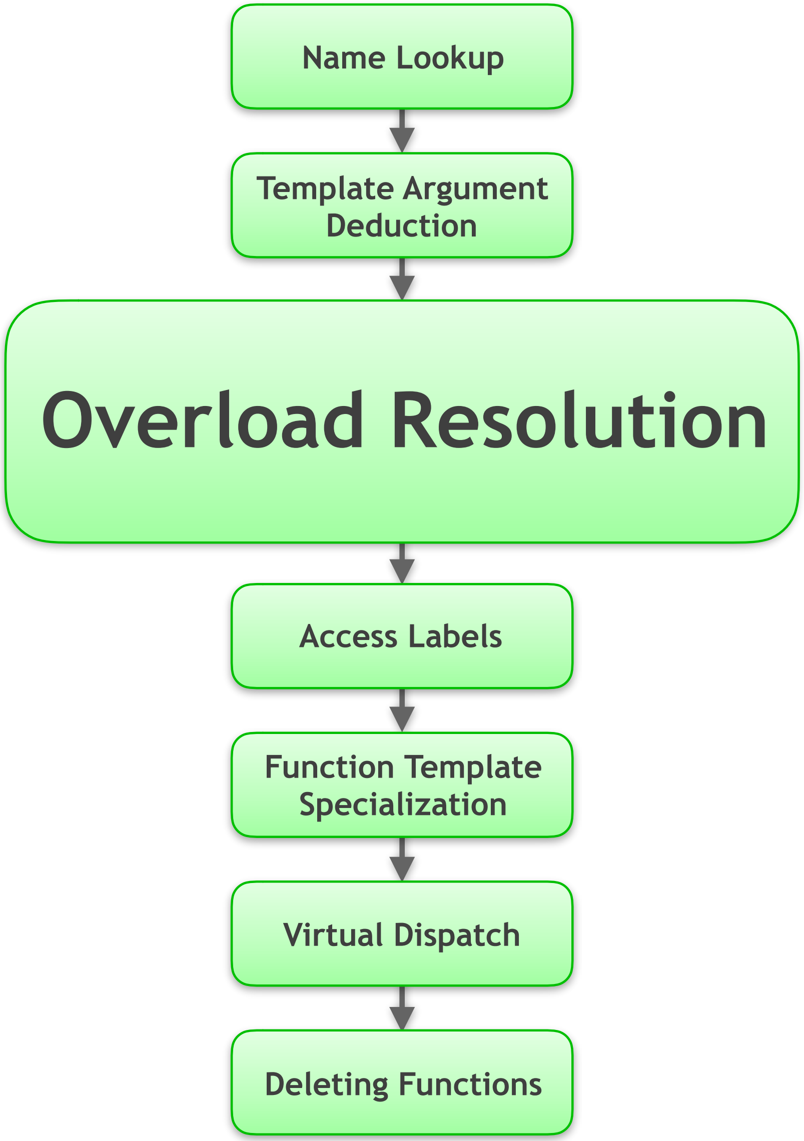
Access Labels

Function Template Specialization

Virtual Dispatch

Deleting Functions

Overload Resolution



Name Lookup

Template Argument Deduction

Overload Resolution

Access Labels

Function Template Specialization

Virtual Dispatch

Deleting Functions

Overload Resolution

1. **Compile a list of viable candidates**: Select all functions from the given list of candidates that match the number of given arguments and that could be called (with or without conversion).
2. **Find the best match from the list of viable candidates:** Determine a single function that fits the given arguments best.

(Non-)Viable Candidates

```
struct Widget { Widget( int ); };

// Viable candiates
void f( int );                 // Exact/identity match
void f( const int& );          // Trivial conversions
void f( double );              // Standard conversions
void f( Widget );              // User-defined conversions
void f( int, int = 0 );        // Default arguments
void f( integral auto );       // Matching constraints
void f( ... );                 // Ellipsis argument

// Non-viable candidates
void f();                      // Less parameters than arguments
void f( int, double );         // More parameters than arguments
void f( std::string );         // No conversion available
void f( floating_point auto ); // Violated constraints


int main()
{
   f( 42 );  // Call 'f()' with a single 'int' argument
}
```

Overload Resolution

1. **Compile a list of viable candidates**: Select all functions from the given list of candidates that match the number of given arguments and that could be called (with or without conversion).
2. **Find the best match from the list of viable candidates:** Determine a single function that fits the given arguments best.

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**

10. **Ellipsis argument**

Ranking

```
void f( int& );         // (1) ← Identity match (rank 1)
void f( double );       // (2) ← Standard conversion (rank 3)


int main()
{
   int i = 42;
   f( i );   // Calls (1) (identity match, rank 1)
}
```

Ranking

```
void f( int& );         // (1)  ← Identity match (rank 1)

void f( double );       // (2)  ← Standard conversion (rank 3)

void f( const int& );   // (3)  ← Trivial conversion (rank 1)


int main()
{
   int i = 42;

   f( i );
}
```

Overload resolution

In order to compile a function call, the compiler must first perform name lookup, which, for functions, may involve argument-dependent lookup, and for function templates may be followed by template argument deduction. If these steps produce more than one *candidate function*, then *overload resolution* is performed to select the function that will actually be called.

In general, the candidate function whose parameters match the arguments most closely is the one that is called.

For other contexts where overloaded function names can appear, see Address of an overloaded function.

If a function cannot be selected by overload resolution (e.g. it is a templated entity with a failed constraint), it cannot be named or otherwise used.

Details

Before overload resolution begins, the functions selected by name lookup and template argument deduction are combined to form the set of *candidate functions* (the exact criteria depend on the context in which overload resolution takes place, see below).

If any candidate function is a member function (static or non-static), but not a constructor, it is treated as if it has an extra parameter (*implicit object parameter*) which represents the object for which they are called and appears before the first of the actual parameters.

Similarly, the object on which a member function is being called is prepended to the argument list as the *implied object argument*.

For member functions of class X, the type of the implicit object parameter is affected by cv-qualifications and ref-qualifications of the member function as described in member functions.

The user-defined conversion functions are considered to be members of the *implied object argument* for the purpose of determining the type of the *implicit object parameter*.

The member functions introduced by a using-declaration into a derived class are considered to be members of the derived class for the purpose of defining the type of the *implicit* object parameter.

For the static member functions, the *implicit object parameter* is considered to match any object: its type is not examined and no conversion sequence is attempted for it.

For the rest of overload resolution, the *implied object argument* is indistinguishable from other arguments, but the following special rules apply to the *implicit object parameter*:

1) user-defined conversions cannot be applied to the implicit object parameter

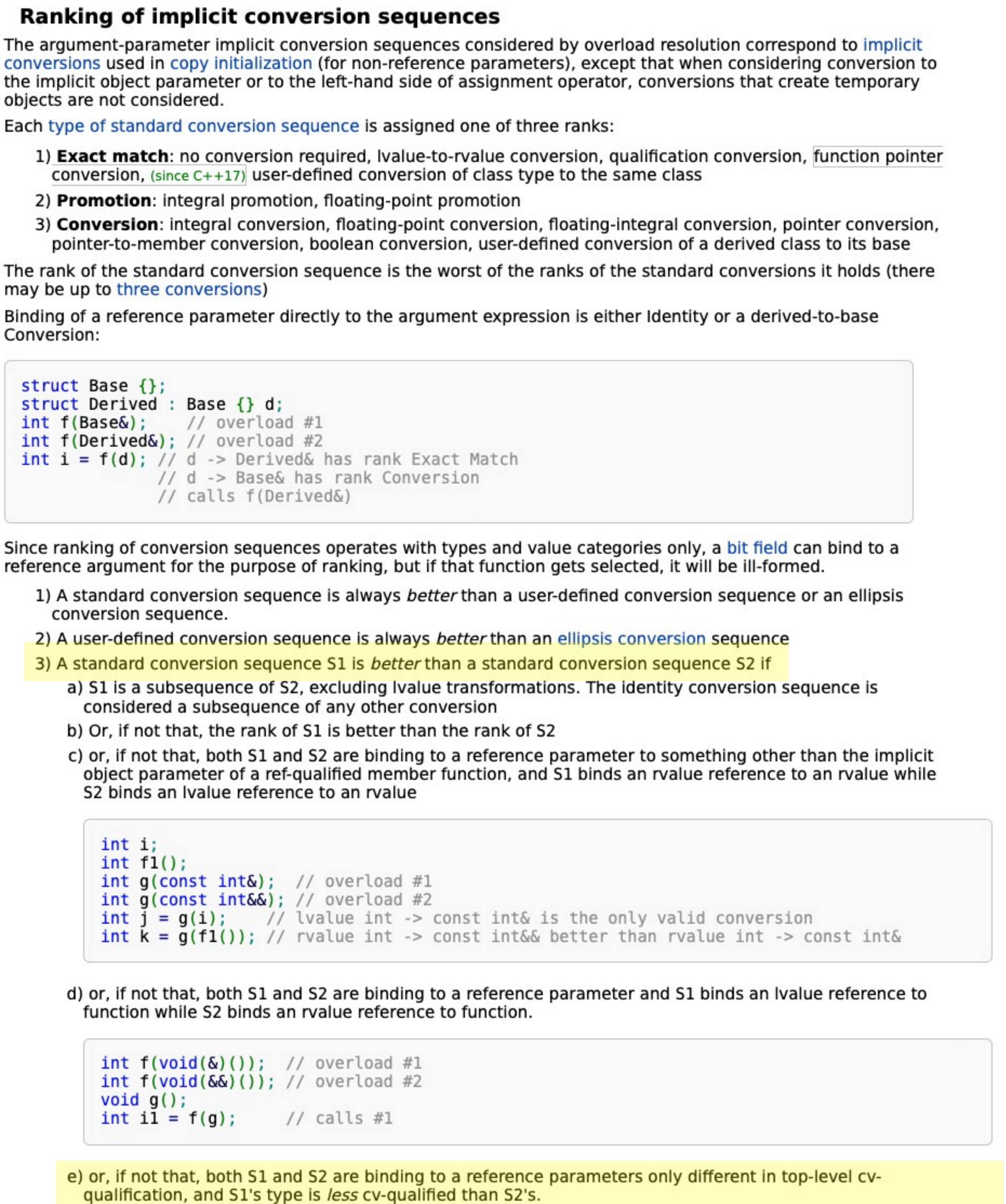
2) rvalues can be bound to non-const implicit object parameter (unless this is for a ref-qualified member function) (since C++11) and do not affect the ranking of the implicit conversions.

```
struct B { void f(int); };
struct A { operator B&(); };
A a;
a.B::f(1); // Error: user-defined conversions cannot be applied
           // to the implicit object parameter
static_cast<B&>(a).f(1); // OK
```

If any candidate is a function template, its specializations are generated using template argument deduction, and such specializations are treated just like non-template functions except where specified otherwise in the tie-breaker rules. If a name refers to one or more function templates and also to a set of overloaded non-template functions, those functions and the specializations generated from the templates are all candidates.

If a constructor template or conversion function template has an conditional explicit specifier which happens to be value-dependent, after deduction, if the context requires a candidate that is not explicit and the generated specialization is explicit, it is removed from the candidate set (since C++20)

Defaulted move constructor and move assignment that are defined as deleted are never included in the list of



Ranking of implicit conversion sequences

The argument-parameter implicit conversion sequences considered by overload resolution correspond to implicit conversions used in copy initialization (for non-reference parameters), except that when considering conversion to the implicit object parameter or to the left-hand side of assignment operator, conversions that create temporary objects are not considered.

Each type of standard conversion sequence is assigned one of three ranks:

1) **Exact match**: no conversion required, lvalue-to-rvalue conversion, qualification conversion, function pointer conversion, (since C++17) user-defined conversion of class type to the same class

2) **Promotion**: integral promotion, floating-point promotion

3) **Conversion**: integral conversion, floating-point conversion, floating-integral conversion, pointer conversion, pointer-to-member conversion, boolean conversion, user-defined conversion of a derived class to its base

The rank of the standard conversion sequence is the worst of the ranks of the standard conversions it holds (there may be up to three conversions)

Binding of a reference parameter directly to the argument expression is either Identity or a derived-to-base Conversion:

```
struct Base {};
struct Derived : Base {} d;
int f(Base&);    // overload #1
int f(Derived&); // overload #2
int i = f(d); // d -> Derived& has rank Exact Match
              // d -> Base& has rank Conversion
              // calls f(Derived&)
```

Since ranking of conversion sequences operates with types and value categories only, a bit field can bind to a reference argument for the purpose of ranking, but if that function gets selected, it will be ill-formed.

1) A standard conversion sequence is always *better* than a user-defined conversion sequence or an ellipsis conversion sequence.

2) A user-defined conversion sequence is always *better* than an ellipsis conversion sequence

3) A standard conversion sequence S1 is *better* than a standard conversion sequence S2 if

a) S1 is a subsequence of S2, excluding lvalue transformations. The identity conversion sequence is considered a subsequence of any other conversion

b) Or, if not that, the rank of S1 is better than the rank of S2

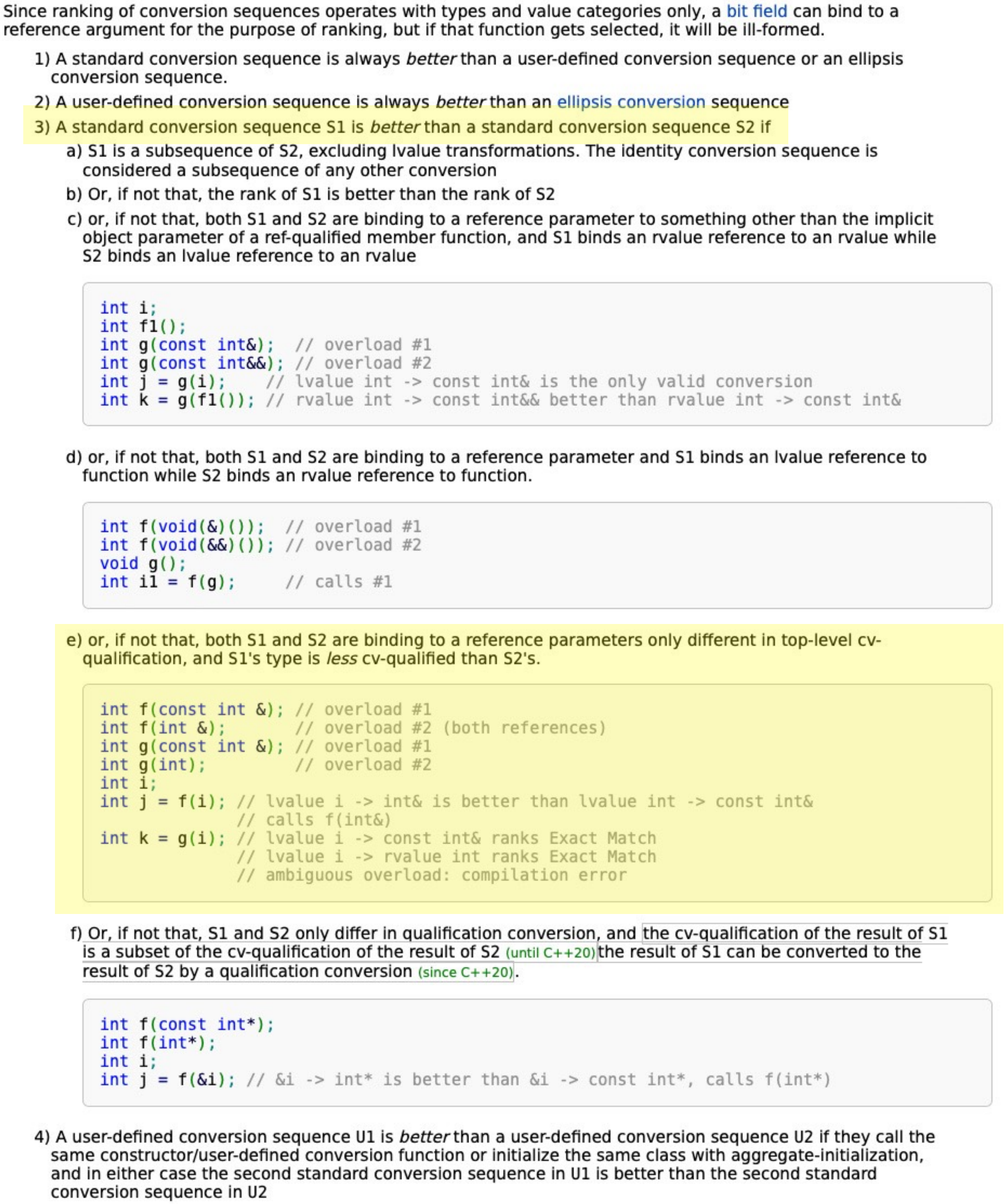
c) or, if not that, both S1 and S2 are binding to a reference parameter to something other than the implicit object parameter of a ref-qualified member function, and S1 binds an rvalue reference to an rvalue while S2 binds an lvalue reference to an rvalue

```
int i;
int f1();
int g(const int&);  // overload #1
int g(const int&&); // overload #2
int j = g(i);    // lvalue int -> const int& is the only valid conversion
int k = g(f1()); // rvalue int -> const int&& better than rvalue int -> const int&
```

d) or, if not that, both S1 and S2 are binding to a reference parameter and S1 binds an lvalue reference to function while S2 binds an rvalue reference to function.

```
int f(void(&)());  // overload #1
int f(void(&&)()); // overload #2
void g();
int i1 = f(g);     // calls #1
```

e) or, if not that, both S1 and S2 are binding to a reference parameters only different in top-level cv-qualification, and S1's type is *less* cv-qualified than S2's.



Since ranking of conversion sequences operates with types and value categories only, a bit field can bind to a reference argument for the purpose of ranking, but if that function gets selected, it will be ill-formed.

1) A standard conversion sequence is always *better* than a user-defined conversion sequence or an ellipsis conversion sequence.

2) A user-defined conversion sequence is always *better* than an ellipsis conversion sequence

3) A standard conversion sequence S1 is *better* than a standard conversion sequence S2 if

a) S1 is a subsequence of S2, excluding lvalue transformations. The identity conversion sequence is considered a subsequence of any other conversion

b) Or, if not that, the rank of S1 is better than the rank of S2

c) or, if not that, both S1 and S2 are binding to a reference parameter to something other than the implicit object parameter of a ref-qualified member function, and S1 binds an rvalue reference to an rvalue while S2 binds an lvalue reference to an rvalue

```
int i;
int f1();
int g(const int&);  // overload #1
int g(const int&&); // overload #2
int j = g(i);    // lvalue int -> const int& is the only valid conversion
int k = g(f1()); // rvalue int -> const int&& better than rvalue int -> const int&
```

d) or, if not that, both S1 and S2 are binding to a reference parameter and S1 binds an lvalue reference to function while S2 binds an rvalue reference to function.

```
int f(void(&)());  // overload #1
int f(void(&&)()); // overload #2
void g();
int i1 = f(g);     // calls #1
```

e) or, if not that, both S1 and S2 are binding to a reference parameters only different in top-level cv-qualification, and S1's type is *less* cv-qualified than S2's.

```
int f(const int &); // overload #1
int f(int &);       // overload #2 (both references)
int g(const int &); // overload #1
int g(int);         // overload #2
int i;
int j = f(i); // lvalue i -> int& is better than lvalue int -> const int&
              // calls f(int&)
int k = g(i); // lvalue i -> const int& ranks Exact Match
              // lvalue i -> rvalue int ranks Exact Match
              // ambiguous overload: compilation error
```

f) Or, if not that, S1 and S2 only differ in qualification conversion, and the cv-qualification of the result of S1 is a subset of the cv-qualification of the result of S2 (until C++20) the result of S1 can be converted to the result of S2 by a qualification conversion (since C++20).

```
int f(const int*);
int f(int*);
int i;
int j = f(&i); // &i -> int* is better than &i -> const int*, calls f(int*)
```

4) A user-defined conversion sequence U1 is *better* than a user-defined conversion sequence U2 if they call the same constructor/user-defined conversion function or initialize the same class with aggregate-initialization, and in either case the second standard conversion sequence in U1 is better than the second standard conversion sequence in U2

Ranking

```
void f( int& );          // (1)  ← Identity match (rank 1)

void f( double );        // (2)  ← Standard conversion (rank 3)

void f( const int& );    // (3)  ← Trivial conversion (rank 1)


int main()
{
   int i = 42;

   f( i );   // Calls (1) (identity match, rank 1)
}
```

Ambiguous Function Calls

If the compiler cannot find a best match, the call is ambiguous.

```
void f( float  );    // (1) ← Standard conversion (rank 3)

void f( double );    // (2) ← Different standard
                                conversion (rank 3)


int main()
{
   int i = 42;

   f( i );  // Ambiguous function call: same rank, no special rule
}
```

Templates vs. Non-Templates

```cpp
void f( int );            // (1)

void f( integral auto );  // (2)


int main()
{
   int i = 42;

   f( i );
}
```

Templates vs. Non-Templates

```
void f( int );            // (1) ← Identity match (rank 1)
void f( integral auto );  // (2)
template<> void f( int ); // (2) ← Also identity match (rank 1)

int main()
{
   int i = 42;
   f( i );
}
```

Overload resolution

In order to compile a function call, the compiler must first perform name lookup, which, for functions, may involve argument-dependent lookup, and for function templates may be followed by template argument deduction. If these steps produce more than one *candidate function*, then *overload resolution* is performed to select the function that will actually be called.

In general, the candidate function whose parameters match the arguments most closely is the one that is called.

For other contexts where overloaded function names can appear, see Address of an overloaded function.

If a function cannot be selected by overload resolution (e.g. it is a templated entity with a failed constraint), it cannot be named or otherwise used.

Details

Before overload resolution begins, the functions selected by name lookup and template argument deduction are combined to form the set of *candidate functions* (the exact criteria depend on the context in which overload resolution takes place, see below).

If any candidate function is a member function (static or non-static), but not a constructor, it is treated as if it has an extra parameter (*implicit object parameter*) which represents the object for which they are called and appears before the first of the actual parameters.

Similarly, the object on which a member function is being called is prepended to the argument list as the *implied object argument*.

For member functions of class X, the type of the implicit object parameter is affected by cv-qualifications and ref-qualifications of the member function as described in member functions.

The user-defined conversion functions are considered to be members of the *implied object argument* for the purpose of determining the type of the *implicit object parameter*.

The member functions introduced by a using-declaration into a derived class are considered to be members of the derived class for the purpose of defining the type of the *implicit* object parameter.

For the static member functions, the *implicit object parameter* is considered to match any object: its type is not examined and no conversion sequence is attempted for it.

For the rest of overload resolution, the *implied object argument* is indistinguishable from other arguments, but the following special rules apply to the *implicit object parameter*:

1) user-defined conversions cannot be applied to the implicit object parameter

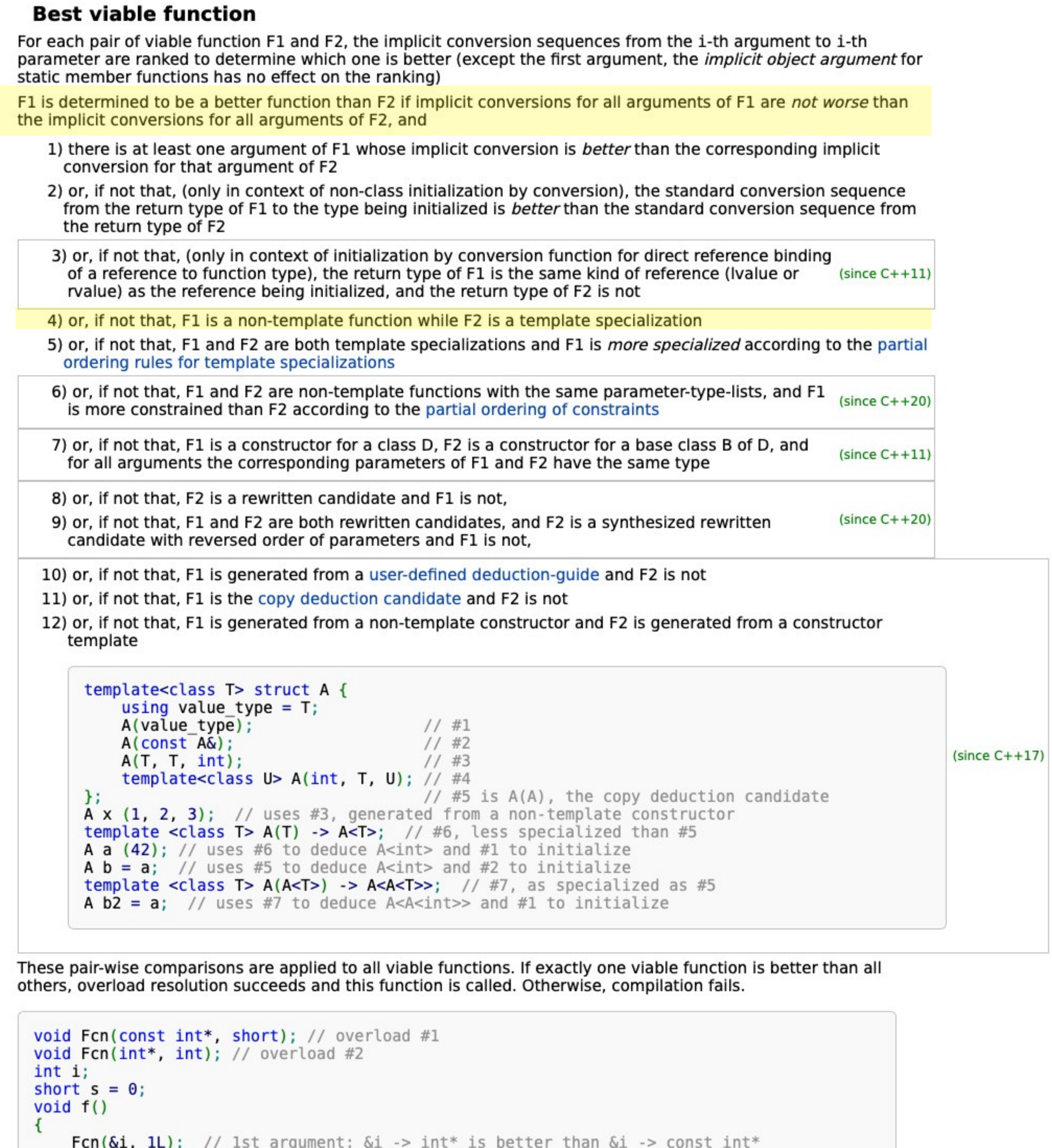
2) rvalues can be bound to non-const implicit object parameter (unless this is for a ref-qualified member function) (since C++11) and do not affect the ranking of the implicit conversions.

```
struct B { void f(int); };
struct A { operator B&(); };
A a;
a.B::f(1); // Error: user-defined conversions cannot be applied
           // to the implicit object parameter
static_cast<B&>(a).f(1); // OK
```

If any candidate is a function template, its specializations are generated using template argument deduction, and such specializations are treated just like non-template functions except where specified otherwise in the tie-breaker rules. If a name refers to one or more function templates and also to a set of overloaded non-template functions, those functions and the specializations generated from the templates are all candidates.

If a constructor template or conversion function template has an conditional explicit specifier which happens to be value-dependent, after deduction, if the context requires a candidate that is not explicit and the generated specialization is explicit, it is removed from the candidate set (since C++20)

Defaulted move constructor and move assignment that are defined as deleted are never included in the list of



Best viable function

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 (except the first argument, the *implicit object argument* for static member functions has no effect on the ranking)

F1 is determined to be a better function than F2 if implicit conversions for all arguments of F1 are *not worse* than the implicit conversions for all arguments of F2, and

1) there is at least one argument of F1 whose implicit conversion is *better* than the corresponding implicit conversion for that argument of F2

2) or, if not that, (only in context of non-class initialization by conversion), the standard conversion sequence from the return type of F1 to the type being initialized is *better* than the standard conversion sequence from the return type of F2

3) or, if not that, (only in context of initialization by conversion function for direct reference binding of a reference to function type), the return type of F1 is the same kind of reference (lvalue or rvalue) as the reference being initialized, and the return type of F2 is not (since C++11)

4) or, if not that, F1 is a non-template function while F2 is a template specialization

5) or, if not that, F1 and F2 are both template specializations and F1 is *more specialized* according to the partial ordering rules for template specializations

6) or, if not that, F1 and F2 are non-template functions with the same parameter-type-lists, and F1 is more constrained than F2 according to the partial ordering of constraints (since C++20)

7) or, if not that, F1 is a constructor for a class D, F2 is a constructor for a base class B of D, and for all arguments the corresponding parameters of F1 and F2 have the same type (since C++11)

8) or, if not that, F2 is a rewritten candidate and F1 is not,

9) or, if not that, F1 and F2 are both rewritten candidates, and F2 is a synthesized rewritten candidate with reversed order of parameters and F1 is not, (since C++20)

10) or, if not that, F1 is generated from a user-defined deduction-guide and F2 is not

11) or, if not that, F1 is the copy deduction candidate and F2 is not

12) or, if not that, F1 is generated from a non-template constructor and F2 is generated from a constructor template

```
template<class T> struct A {
    using value_type = T;
    A(value_type);                   // #1
    A(const A&);                     // #2
    A(T, T, int);                    // #3
    template<class U> A(int, T, U); // #4
};                                   // #5 is A(A), the copy deduction candidate
A x (1, 2, 3);  // uses #3, generated from a non-template constructor
template <class T> A(T) -> A<T>;  // #6, less specialized than #5
A a (42); // uses #6 to deduce A<int> and #1 to initialize
A b = a;  // uses #5 to deduce A<int> and #2 to initialize
template <class T> A(A<T>) -> A<A<T>>;  // #7, as specialized as #5
A b2 = a;  // uses #7 to deduce A<A<int>> and #1 to initialize
```

(since C++17)

These pair-wise comparisons are applied to all viable functions. If exactly one viable function is better than all others, overload resolution succeeds and this function is called. Otherwise, compilation fails.

```
void Fcn(const int*, short); // overload #1
void Fcn(int*, int); // overload #2
int i;
short s = 0;
void f()
{
    Fcn(&i, 1L);  // 1st argument: &i -> int* is better than &i -> const int*
```

Templates vs. Non-Templates

```
void f( int );                  // (1) ← Identity match (rank 1)

void f( integral auto );   // (2)

template<> void f( int ); // (2) ← Also identity match (rank 1)


int main()
{
   int i = 42;

   f( i ); // Calls (1) (identity match, rank 1)
}
```

Promotions (Rank 2)

The following standard conversions count as promotions:

- Integral promotion
  - `unsigned short` ➜ `unsigned int` or `int` (depending on your platform)
  - `short` ➜ `int`
  - `char` ➜ `int` or `unsigned int` (depending on your platform)
  - `bool` to `int` (0 or 1)
- Floating point promotion
  - `float` ➜ `double`

Promotions (Rank 2)

```
void f( int    );   // (1) ← Integral promotion (rank 2)
void f( double );   // (2) ← Standard conversion (rank 3)


int main()
{
   short v = 42;
   f( v );  // Calls (1) (integral promotion, rank 2)
}
```

Promotions (Rank 2)

```
void f( int    );   // (1) ←── Standard conversion (rank 3)

void f( double );   // (2) ←── Floating point promotion
                               (rank 2)


int main()
{
   float v = 4.2F;

   f( v );  // Calls (2) (floating point promotion, rank 2)
}
```

Standard Conversions (Rank 3)

A standard conversion is when the compiler transforms a fundamental / built-in data type into a related fundamental/built-in data type:

- Integral conversion
- Floating point conversion
- Integral to floating point conversion
- Floating point to integral conversion
- Pointer conversion
  - Derived to base class conversions (`Sheep*` ➜ `Animal*`)
  - Conversions to `void*`
- `bool` conversion
  - expressions converted to `bool`

Standard Conversions (Rank 3)

```
void f( double );   // (1) ← Standard conversion (rank 3)



int main()
{
   unsigned int ui = 42U;
   f( ui );  // Calls (1) (standard conversion, rank 3)
}
```

Standard Conversions (Rank 3)

```
void f( int );     // (1) ← Standard conversion (rank 3)


int main()
{
   unsigned int ui = 42U;
   f( ui );  // Calls (1) (standard conversion, rank 3)
}
```

Standard Conversions (Rank 3)

```
void f( double );   // (1) ←— Standard conversion (rank 3)
void f( int );      // (2) ←— Standard conversion (rank 3)


int main()
{
   unsigned int ui = 42U;
   f( ui );  // Ambiguous function call: same rank, no special rule
}
```

Standard Conversions (Rank 3)

```
struct A {};
struct B : public A {};
struct C : public B {};

void f( A* );  // (1)  ←  Pointer conversion (rank 3)
void f( B* );  // (2)  ←  Also pointer conversion (rank 3)

int main()
{
   C c{};
   f( &c );  // Calls (2) (closest pointer conversion, rank 3)
}
```

User-Defined Conversions

A user-defined conversion is provided via constructors or conversion operators:

```
struct Widget
{
   // Constructors
   Widget( int );      // Conversion from ‘int' to 'Widget'
   Widget( double );   // Conversion from 'double' to 'Widget'

   // Conversion operators
   operator int();     // Conversion from 'Widget' to ‘int'
   operator double();  // Conversion from 'Widget' to ‘double'
};
```

User-Defined Conversions

```
struct Widget { Widget( int ); };

void f( long );    // (1) ← Standard conversion (rank 3)
void f( Widget );  // (2) ← User-defined conversion


int main()
{
   int i = 42;
   f( i );   // Calls (1) (standard conversion, rank 3)
}
```

Ranking of implicit conversion sequences

The argument-parameter implicit conversion sequences considered by overload resolution correspond to implicit conversions used in copy initialization (for non-reference parameters), except that when considering conversion to the implicit object parameter or to the left-hand side of assignment operator, conversions that create temporary objects are not considered.

Each type of standard conversion sequence is assigned one of three ranks:

1) **Exact match**: no conversion required, lvalue-to-rvalue conversion, qualification conversion, function pointer conversion, (since C++17) user-defined conversion of class type to the same class
2) **Promotion**: integral promotion, floating-point promotion
3) **Conversion**: integral conversion, floating-point conversion, floating-integral conversion, pointer conversion, pointer-to-member conversion, boolean conversion, user-defined conversion of a derived class to its base

The rank of the standard conversion sequence is the worst of the ranks of the standard conversions it holds (there may be up to three conversions)

Binding of a reference parameter directly to the argument expression is either Identity or a derived-to-base Conversion:

```
struct Base {};
struct Derived : Base {} d;
int f(Base&);    // overload #1
int f(Derived&); // overload #2
int i = f(d); // d -> Derived& has rank Exact Match
              // d -> Base& has rank Conversion
              // calls f(Derived&)
```

Since ranking of conversion sequences operates with types and value categories only, a bit field can bind to a reference argument for the purpose of ranking, but if that function gets selected, it will be ill-formed.

1) A standard conversion sequence is always *better* than a user-defined conversion sequence or an ellipsis conversion sequence.
2) A user-defined conversion sequence is always *better* than an ellipsis conversion sequence
3) A standard conversion sequence S1 is *better* than a standard conversion sequence S2 if
   a) S1 is a subsequence of S2, excluding lvalue transformations. The identity conversion sequence is considered a subsequence of any other conversion
   b) Or, if not that, the rank of S1 is better than the rank of S2
   c) or, if not that, both S1 and S2 are binding to a reference parameter to something other than the implicit object parameter of a ref-qualified member function, and S1 binds an rvalue reference to an rvalue while S2 binds an lvalue reference to an rvalue

```
int i;
int f1();
int g(const int&);  // overload #1
int g(const int&&); // overload #2
int j = g(i);    // lvalue int -> const int& is the only valid conversion
int k = g(f1()); // rvalue int -> const int&& better than rvalue int -> const int&
```

   d) or, if not that, both S1 and S2 are binding to a reference parameter and S1 binds an lvalue reference to function while S2 binds an rvalue reference to function.

```
int f(void(&)());  // overload #1
int f(void(&&)()); // overload #2
void g();
int i1 = f(g);     // calls #1
```

   e) or, if not that, both S1 and S2 are binding to a reference parameters only different in top-level cv-qualification, and S1's type is *less* cv-qualified than S2's.

User-Defined Conversions

```
struct Widget {
   Widget( int );      // (1)  ← User-defined conversion

   Widget( double );   // (2)  ← User-defined + standard
};                                conversion

void f( Widget );

int main()
{
   int i = 42;
   f( i );   // Calls (1) (user-defined conversion)
}
```

User-Defined Conversions

```
struct Widget {
   Widget( int );      // (1)  ← User-defined conversion +
                                 promotion

   Widget( double );   // (2)  ← User-defined + standard
};                               conversion


void f( Widget );


int main()
{
   short s = 42;

   f( s );   // Calls (1) (user-defined conversion + promotion)
}
```

Ranking of implicit conversion sequences

The argument-parameter implicit conversion sequences considered by overload resolution correspond to implicit conversions used in copy initialization (for non-reference parameters), except that when considering conversion to the implicit object parameter or to the left-hand side of assignment operator, conversions that create temporary objects are not considered.

Each type of standard conversion sequence is assigned one of three ranks:

1) **Exact match**: no conversion required, lvalue-to-rvalue conversion, qualification conversion, function pointer conversion, (since C++17) user-defined conversion of class type to the same class
2) **Promotion**: integral promotion, floating-point promotion
3) **Conversion**: integral conversion, floating-point conversion, floating-integral conversion, pointer conversion, pointer-to-member conversion, boolean conversion, user-defined conversion of a derived class to its base

The rank of the standard conversion sequence is the worst of the ranks of the standard conversions it holds (there may be up to three conversions)

Binding of a reference parameter directly to the argument expression is either Identity or a derived-to-base Conversion:

```
struct Base {};
struct Derived : Base {} d;
int f(Base&);    // overload #1
int f(Derived&); // overload #2
int i = f(d); // d -> Derived& has rank Exact Match
              // d -> Base& has rank Conversion
              // calls f(Derived&)
```

Since ranking of conversion sequences operates with types and value categories only, a bit field can bind to a reference argument for the purpose of ranking, but if that function gets selected, it will be ill-formed.

1) A standard conversion sequence is always *better* than a user-defined conversion sequence or an ellipsis conversion sequence.
2) A user-defined conversion sequence is always *better* than an ellipsis conversion sequence
3) A standard conversion sequence S1 is *better* than a standard conversion sequence S2 if
   a) S1 is a subsequence of S2, excluding lvalue transformations. The identity conversion sequence is considered a subsequence of any other conversion
   b) Or, if not that, the rank of S1 is better than the rank of S2
   c) or, if not that, both S1 and S2 are binding to a reference parameter to something other than the implicit object parameter of a ref-qualified member function, and S1 binds an rvalue reference to an rvalue while S2 binds an lvalue reference to an rvalue

```
int i;
int f1();
int g(const int&);  // overload #1
int g(const int&&); // overload #2
int j = g(i);    // lvalue int -> const int& is the only valid conversion
int k = g(f1()); // rvalue int -> const int&& better than rvalue int -> const int&
```

   d) or, if not that, both S1 and S2 are binding to a reference parameter and S1 binds an lvalue reference to function while S2 binds an rvalue reference to function.

```
int f(void(&)());  // overload #1
int f(void(&&)()); // overload #2
void g();
int i1 = f(g);     // calls #1
```

   e) or, if not that, both S1 and S2 are binding to a reference parameters only different in top-level cv-qualification, and S1's type is *less* cv-qualified than S2's

```
int f(const int &); // overload #1
int f(int &);       // overload #2 (both references)
int g(const int &); // overload #1
int g(int);         // overload #2
int i;
int j = f(i); // lvalue i -> int& is better than lvalue int -> const int&
              // calls f(int&)
int k = g(i); // lvalue i -> const int& ranks Exact Match
              // lvalue i -> rvalue int ranks Exact Match
              // ambiguous overload: compilation error
```

f) Or, if not that, S1 and S2 only differ in qualification conversion, and the cv-qualification of the result of S1 is a subset of the cv-qualification of the result of S2 (until C++20) the result of S1 can be converted to the result of S2 by a qualification conversion (since C++20).

```
int f(const int*);
int f(int*);
int i;
int j = f(&i); // &i -> int* is better than &i -> const int*, calls f(int*)
```

4) A user-defined conversion sequence U1 is *better* than a user-defined conversion sequence U2 if they call the same constructor/user-defined conversion function or initialize the same class with aggregate-initialization, and in either case the second standard conversion sequence in U1 is better than the second standard conversion sequence in U2

```
struct A {
    operator short(); // user-defined conversion function
} a;
int f(int);   // overload #1
int f(float); // overload #2
int i = f(a); // A -> short, followed by short -> int (rank Promotion)
              // A -> short, followed by short -> float (rank Conversion)
              // calls f(int)
```

5) A list-initialization sequence L1 is *better* than list-initialization sequence L2 if L1 initializes an `std::initializer_list` parameter, while L2 does not.

```
void f1(int);                                  // #1
void f1(std::initializer_list<long>);          // #2
void g1() { f1({42}); }                        // chooses #2

void f2(std::pair<const char*, const char*>); // #3
void f2(std::initializer_list<std::string>);  // #4
void g2() { f2({"foo","bar"}); }              // chooses #4
```

6) A list-initialization sequence L1 is *better* than list-initialization sequence L2 if the corresponding parameters are references to arrays, and L1 converts to type "array of N1 T," L2 converts to type "array of N2 T", and N1 is smaller than N2. (since C++14) (until C++20)

6) A list-initialization sequence L1 is *better* than list-initialization sequence L2 if the corresponding parameters are references to arrays, and L1 and L2 convert to arrays of same element type, and either

- the number of elements N1 initialized by L1 is less than the number of elements N2 initialized by L2, or
- N1 is equal to N2 and L2 converts to an array of unknown bound and L1 does not.

```
void f(int    (&&)[] );    // overload #1
void f(double (&&)[] );    // overload #2
void f(int    (&&)[2]);    // overload #3
```

(since C++20)

Finding a Best Match (Multiple Parameters)

For several arguments, the compiler applies the same rules to every single argument. If one function is considered better for at least one argument and equally good for all other arguments, the function is selected as best match. Else the function is ambiguous.

Finding a Best Match (Multiple Parameters)

```
        R1    R2       R3

void f( int, int   , int    );   // (1)

void f( int, double, double );   // (2)
        R1    R3       R3


int main()
{
   f( 1, short(2), 3U );  // Calls (1) (best match)
}
```

Finding a Best Match (Multiple Parameters)

```
        R1    R2      R3

void f( int, int   , int    );    // (1)

void f( int, double, double );    // (2)
        R1    R3      R2


int main()
{
   f( 1, short(2), 3.0F );  // Ambiguous function call
}
```

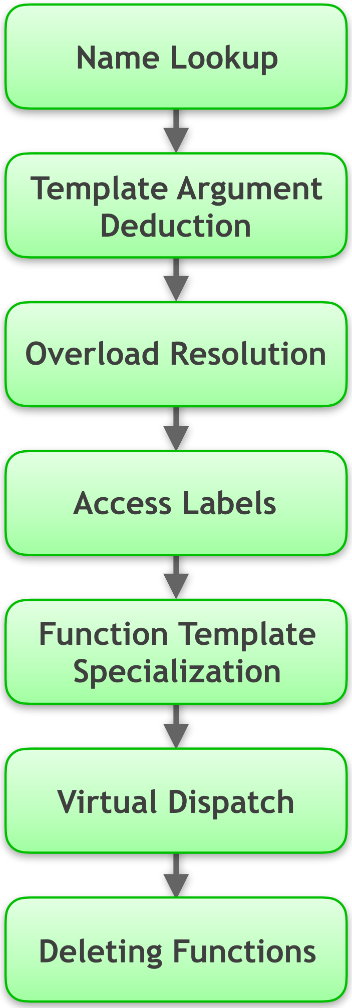
Guidelines

Guideline: Prevent complex overloading situations that may result in surprises during overload resolution.

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

Questions?

Overload Resolution



Name Lookup

Template Argument Deduction

Overload Resolution

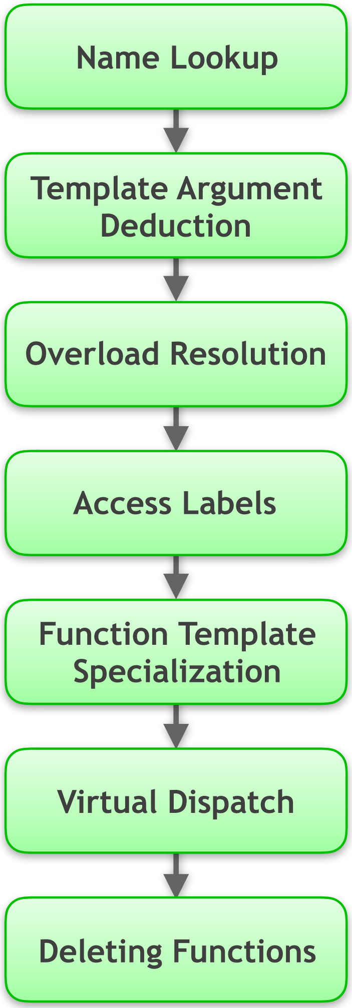
Access Labels

Function Template Specialization

Virtual Dispatch

Deleting Functions

Overload Resolution



Name Lookup

Template Argument Deduction

Overload Resolution

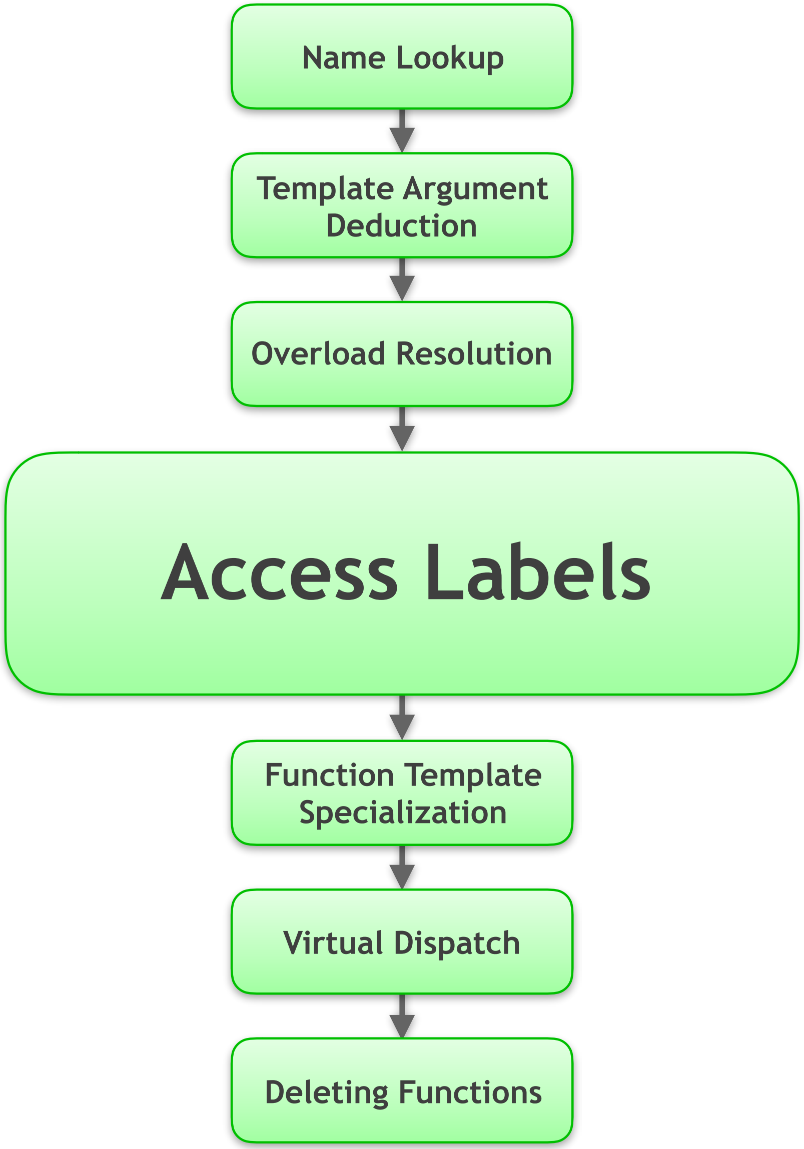
Access Labels

Function Template Specialization

Virtual Dispatch

Deleting Functions

Access Labels



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!
```

Access Labels

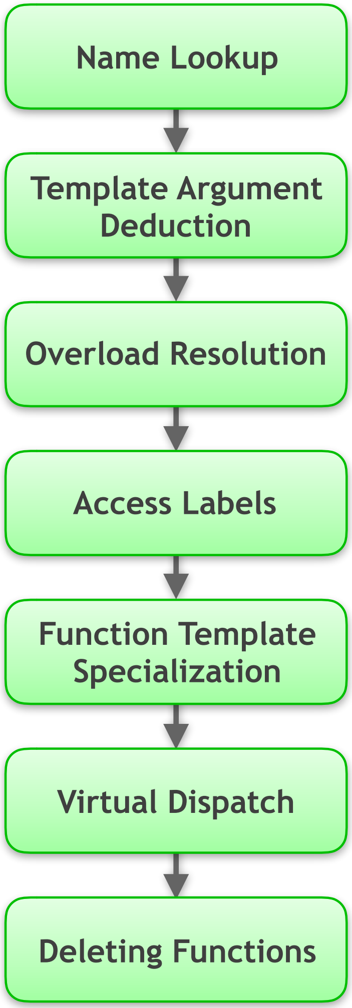
```
glass Object
{
 public:
   void f( int );     // (1)
 private:
   void f( double );  // (2)
};


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

Guidelines

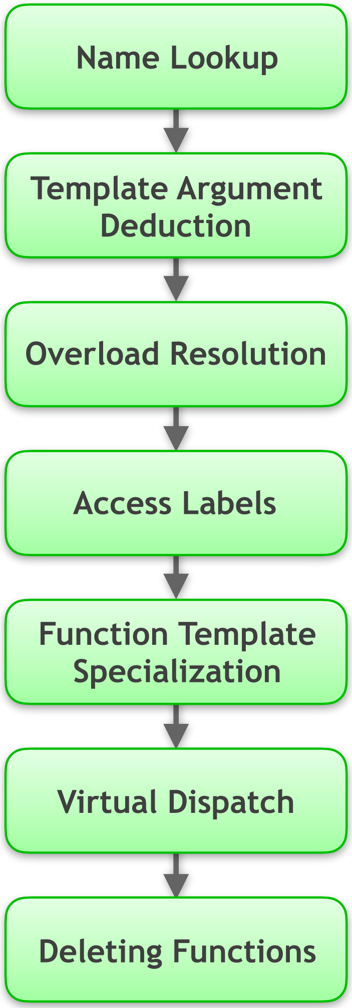
Guideline: Remember that everything inside a class is visible. `public`, `protected` and `private` are merely access labels.

Questions?

Access Labels



Access Labels



Name Lookup

Template Argument Deduction

Overload Resolution

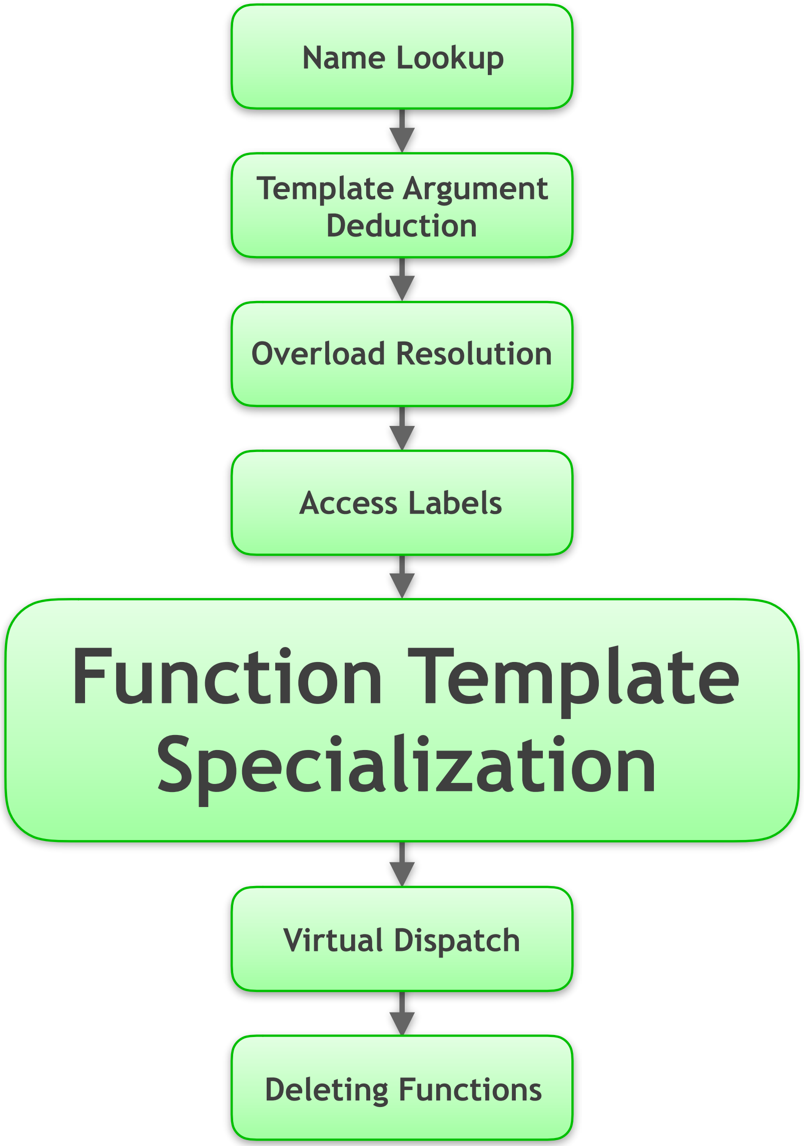
Access Labels

Function Template Specialization

Virtual Dispatch

Deleting Functions

Function Template Specialization



Name Lookup

Template Argument Deduction

Overload Resolution

Access Labels

Function Template Specialization

Virtual Dispatch

Deleting Functions

Function Template Specialization

```
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

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

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

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



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

   f( cp );   // Calls function (2)
}
```

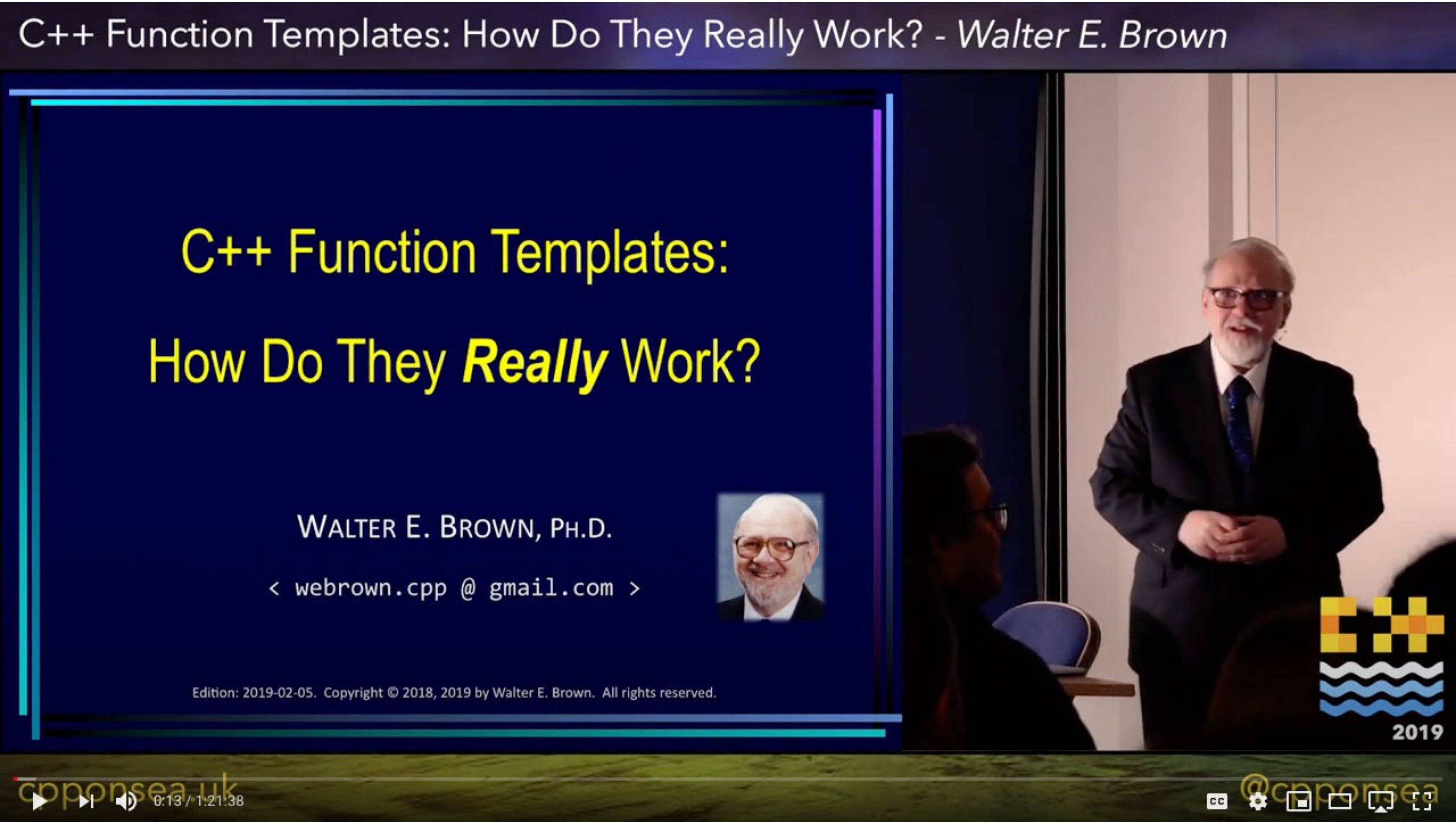
Function Template Specialization

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

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

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



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

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

Guidelines

Guideline: Prefer function overloading to function template specialization.

Function Template Specialization



C++ Function Templates: How Do They Really Work? - *Walter E. Brown*

C++ Function Templates:

How Do They ***Really*** Work?

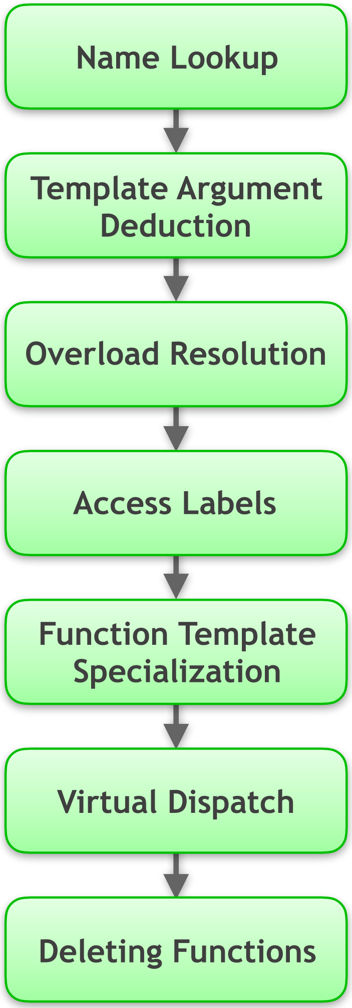
WALTER E. BROWN, PH.D.

< webrown.cpp @ gmail.com >

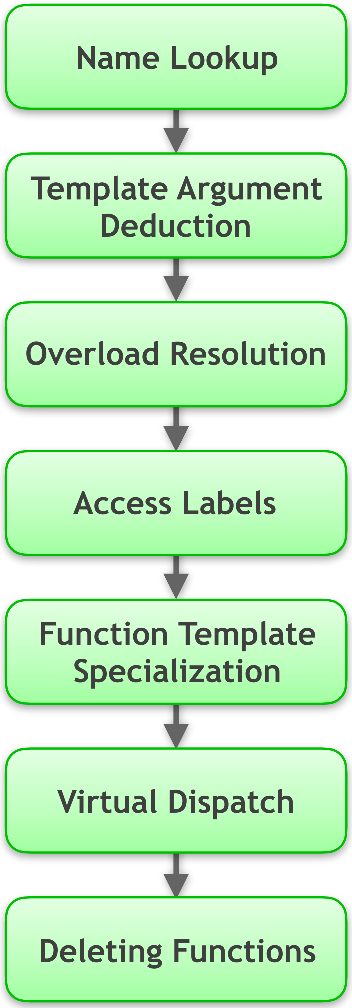
Edition: 2019-02-05. Copyright © 2018, 2019 by Walter E. Brown. All rights reserved.

Questions?

Function Template Specialization



Function Template Specialization


Name Lookup

Template Argument Deduction

Overload Resolution

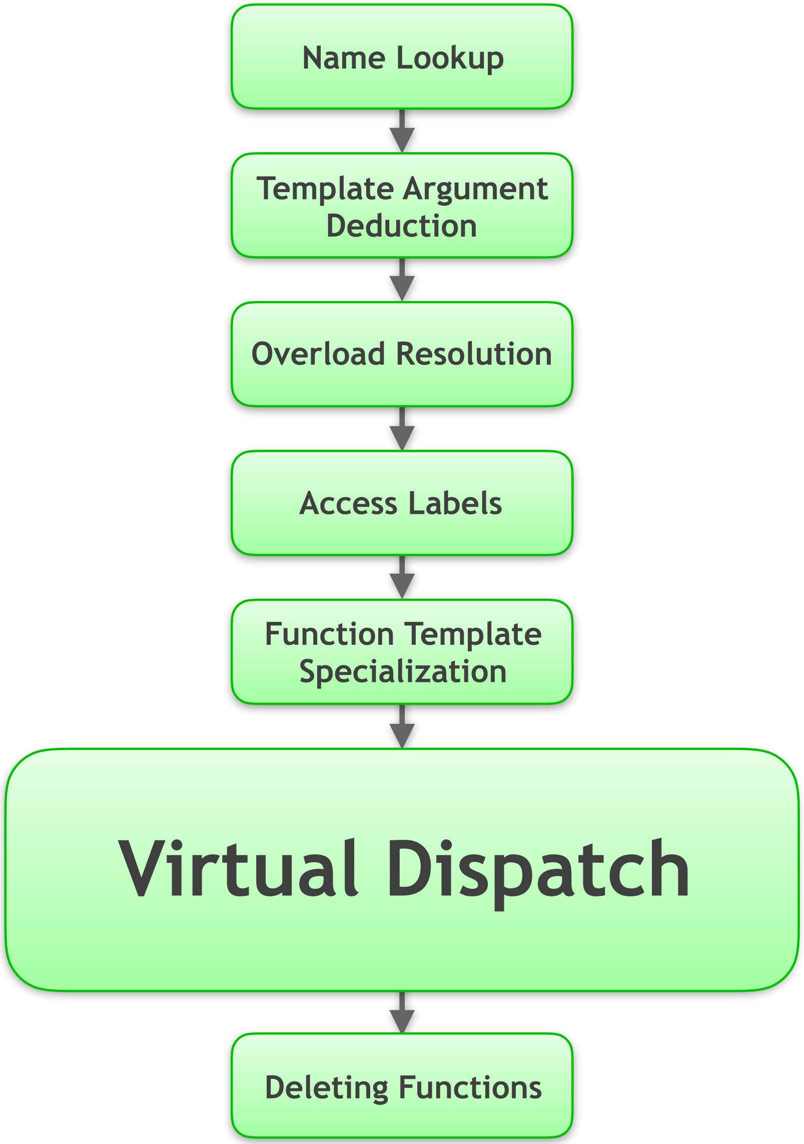
Access Labels

Function Template Specialization

Virtual Dispatch

Deleting Functions

Virtual Dispatch



Name Lookup

Template Argument Deduction

Overload Resolution

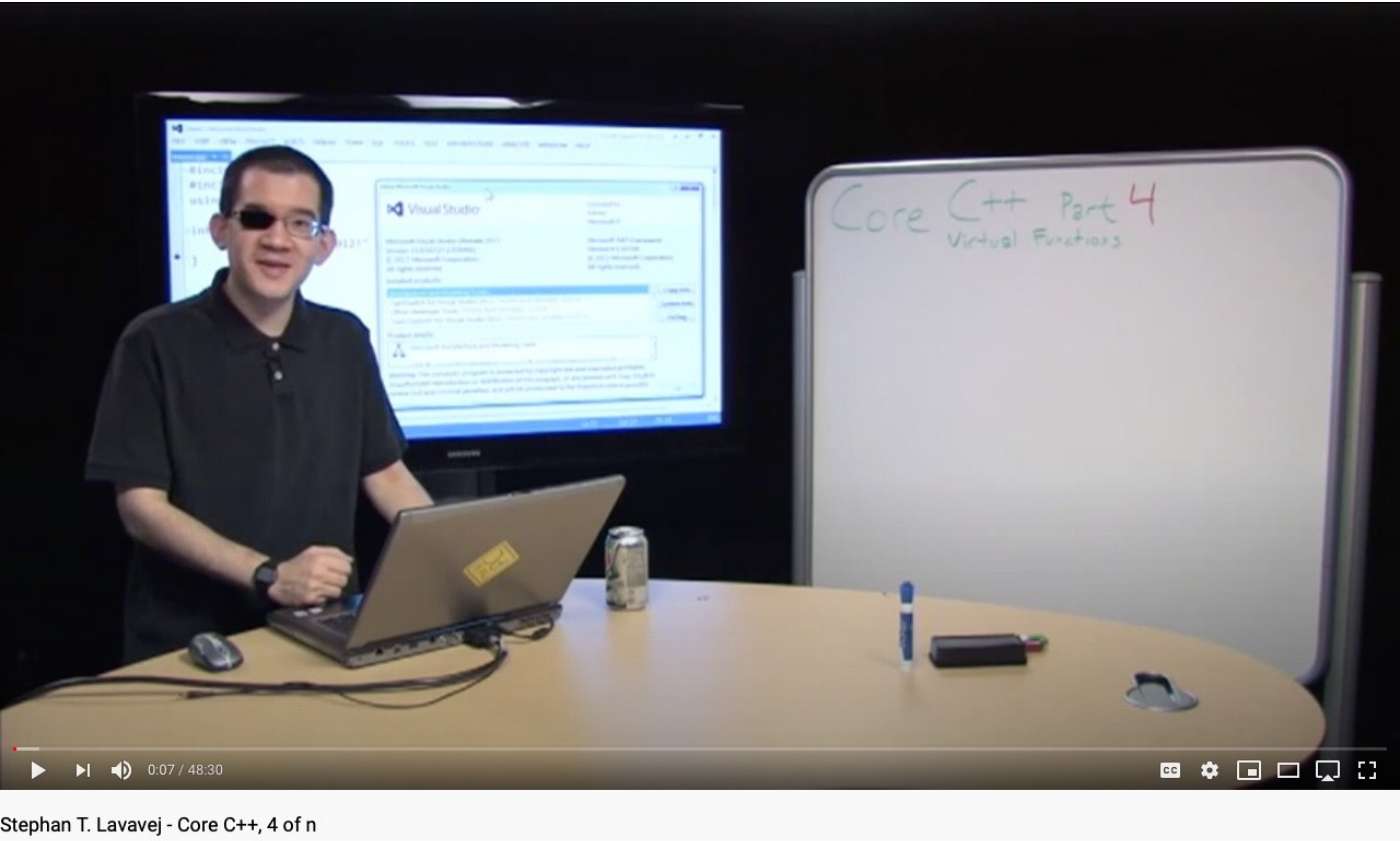
Access Labels

Function Template Specialization

Virtual Dispatch

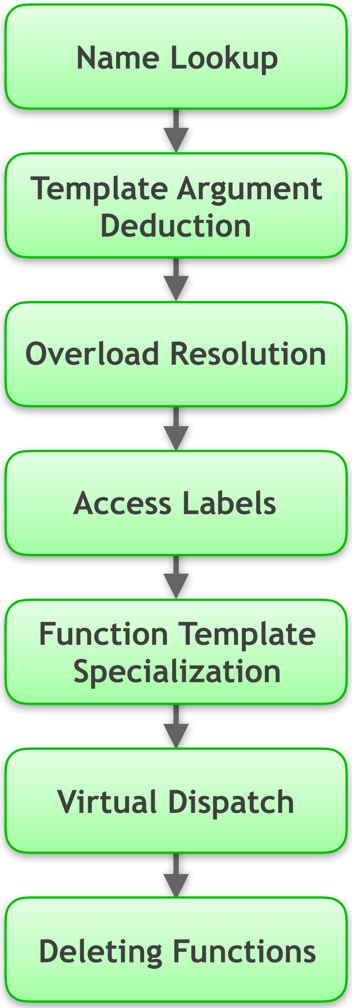
Deleting Functions

Virtual Dispatch

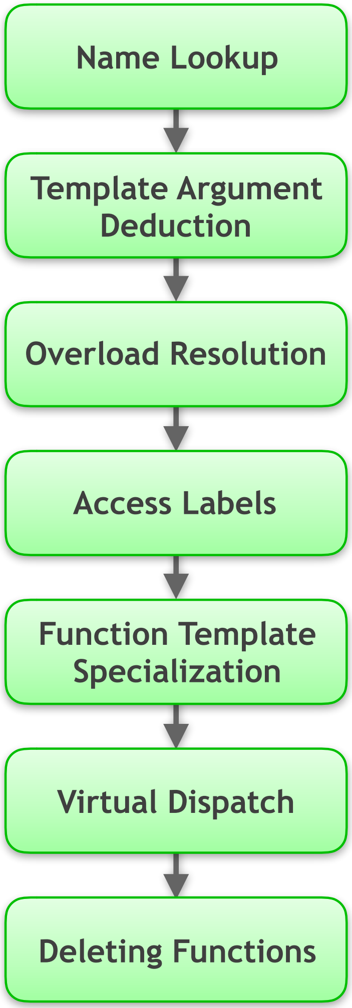


Stephan T. Lavavej - Core C++, 4 of n

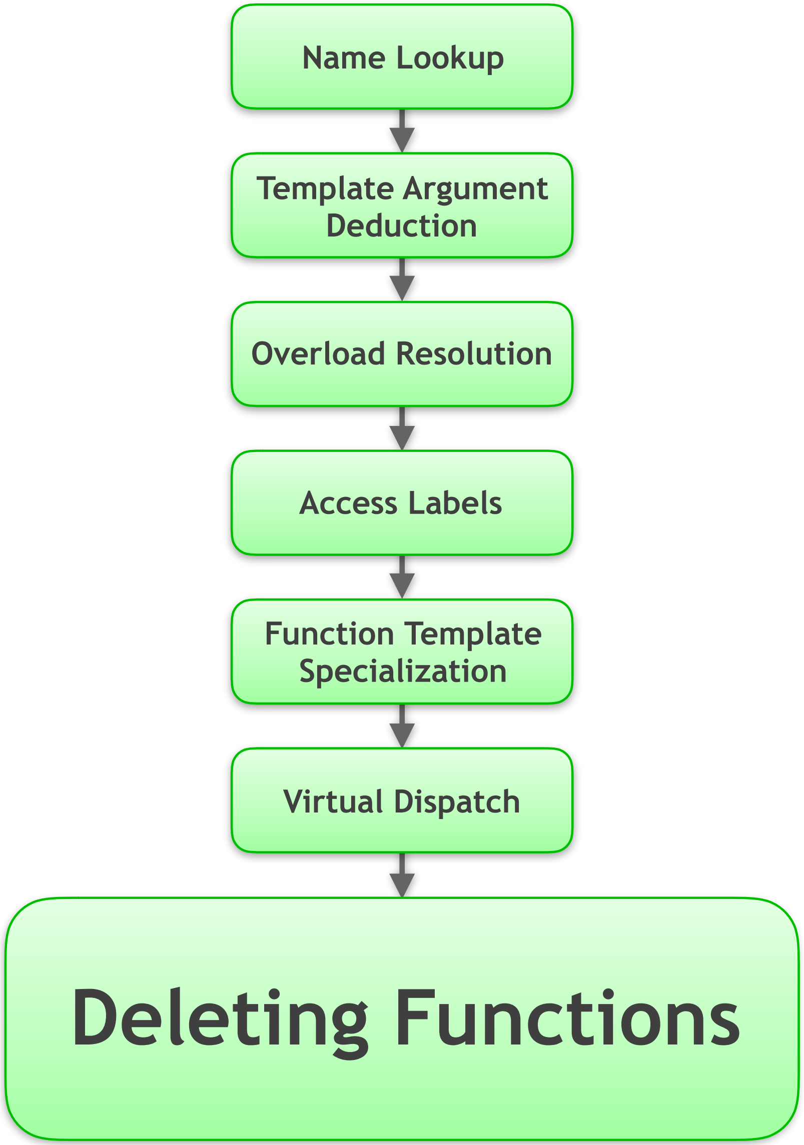
Function Template Specialization



Function Template Specialization



Deleting Functions



Name Lookup

Template Argument Deduction

Overload Resolution

Access Labels

Function Template Specialization

Virtual Dispatch

Deleting Functions

Deleting Functions

```
void f( int );              // (1)

void f( double ) = delete;  // (2)


int main()
{
   f( 42 );  // Calls function (1)

   f( 1.0 ); // Compilation error: Call to deleted function
}
```

Deleting Functions

Guideline: `=delete` doesn’t delete a function, but declares it as uncallable. Any attempt to call the function will result in a compilation error.

Deleting Functions

```
class Widget
{
 public:
   Widget();
   Widget( const Widget& );
   Widget( Widget&& ) = delete;
};


int main()
{
   Widget w1{};                  // Default constructor

   Widget w2( std::move(w1) );  // Compilation error
}
```

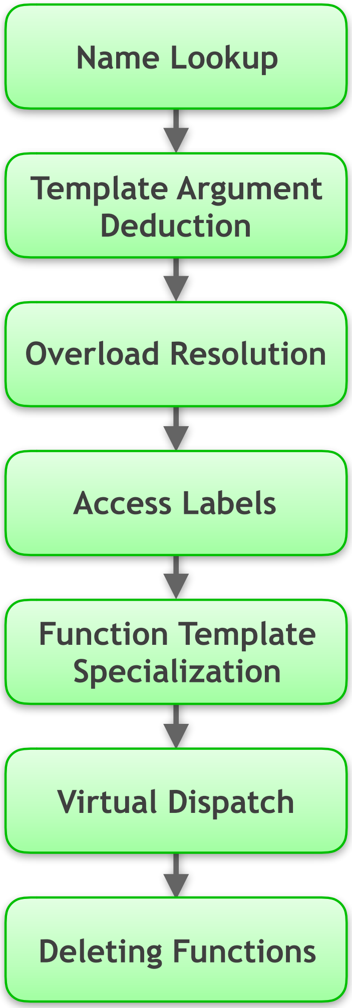
Deleting Functions

```
class Widget
{
 public:
   Widget();
   Widget( const Widget& );

};


int main()
{
   Widget w1{};                    // Default constructor

   Widget w2( std::move(w1) );     // Copy constructor
}
```

Deleting Functions

Core Guideline C.21: If you define or `=delete` any default operation, define or `=delete` them all.

Summary



Name Lookup

Template Argument Deduction

Overload Resolution

Access Labels

Function Template Specialization

Virtual Dispatch

Deleting Functions

Calling Functions

A Tutorial

Klaus Iglberger, Meeting C++ 2020

klaus.iglberger@gmx.de