Back To Basics
Function Call Resolution

BEN SAKS

Cppcon
The C++ Conference

2024
September 15 - 20

Saks & Associates

Saks & Associates
393 Leander Dr.
Springfield, OH 45504-4906 USA
+1-412-521-4117 (voice)
service@saksandassociates.com
saksandassociates.com

2

About Ben Saks

Ben Saks is the chief engineer of Saks & Associates. He is the principal editor and presenter for much of Saks & Associates' training curriculum on the use of C and C++ in embedded systems.

Ben has represented Saks & Associates on the ISO C++ Standards committee as well as two of the committee's study groups:

- SG14 — low-latency
- SG20 — education

Ben has spoken at industry conferences, including *CppCon: The C++ Conference*, the *C++ and System Software Summit*, the *Embedded Systems Conference*, and *NDC Techtown*. At *CppCon*, he's the chair of the Embedded Track and a member of the program committee.

3

More About Ben Saks

Ben previously worked as a software engineer for Vorne Industries, where he used C++ and JavaScript to develop embedded systems that help improve manufacturing productivity in factories all over the world. He is a contributing author on multiple Vorne patents.

Ben earned a B.A. with Distinction in Computer Science from Carleton College.

4

About Dan Saks

Dan Saks is the president of Saks & Associates, which offers training and consulting in C and C++ and their use in developing embedded systems.

Dan wrote the “Programming Pointers” column for *embedded.com* online. He also wrote columns for numerous print publications (when such things existed) including *The C/C++ Users Journal*, *The C++ Report*, *Software Development*, and *Embedded Systems Design*. With Thomas Plum, he wrote *C++ Programming Guidelines*, which won a *1992 Computer Language Magazine Productivity Award*.

Dan has taught C and C++ to thousands of programmers worldwide. He has delivered hundreds of lectures, including a few keynote addresses, at conferences such as the *ACCU (Association of C and C++ Users) Conference, CppCon: The C++ Conference*, *C++ World*, the *Embedded Systems Conference*, *Meeting Embedded*, *NDC Techtown*, and the *Software Development Conference*.

5

More About Dan Saks

Dan served as secretary of the ANSI and ISO C++ Standards committees and as a member of the ANSI C Standards committee. He also contributed to the *CERT Secure C Coding Standard* and the *CERT Secure C++ Coding Standard*.

Dan collaborated with Thomas Plum in writing and maintaining *Suite++™, the Plum Hall Validation Suite for C++*, which tests C++ compilers for conformance with the international standard. He was a Senior Software Engineer for Fischer and Porter (now ABB), where he designed languages and tools for distributed process control. He also worked as a programmer with Sperry Univac (now Unisys).

Dan earned an M.S.E. in Computer Science from the University of Pennsylvania, and a B.S. with Highest Honors in Mathematics/Information Science from Case Western Reserve University.

6

Introduction

- Some C++ language features allow you to create multiple functions with the same name, including:
  - name hiding
  - function overloading
  - function templates
- When a compiler encounters an expression like `f(x, y, z)`, it must consider each of these language features to determine which `f` function to call.

7

Introduction

- When used well, these facilities can produce interfaces that are flexible, easy to use correctly, and hard to use incorrectly.
- When used poorly, the same facilities can produce confusing interfaces that are hard to use correctly.
- To use these facilities effectively, you need a solid grasp of:
  - how they work individually, and
  - how they interact.

8

Introduction

- This session will examine each of these features in turn:
  - Function overloading and overload resolution
  - Name lookup
  - Default function arguments
  - Function templates
- For each feature, we'll explain first how it works on its own, and then how it interacts with the other features.
- We'll close with an example that combines several of these features.

9

Outline

- Function Overloading
- Name Lookup
- Default Function Arguments
- Function Templates
- Tying It All Together

10

Overloading

- A function is ***overloaded*** if there's another function declared:
  - with the same name,
  - in the same scope.
- For example, here's a group of overloaded functions name put:

```
int put(int c);
int put(int c, FILE *f);
int put(char const *s);
int put(char const *s, FILE *f);
```


Overload Resolution

- When the compiler encounters a call to an overloaded function, it selects the function to call using ***overload resolution***.
- The compiler matches:
  - the type(s) of the argument(s) in the call against
  - the type(s) of the parameter(s) in the function declarations.
- For example,

```
                  // calls...
put('a', stdout); // int put(int c, FILE *f);
put("Hello\n");   // int put(char const *s);
```

Overload Resolution

- If the compiler can't find a function that accepts the specified arguments, it issues an error message.
- For example, this is an error:

```
put("n = ", n, stdout);     // compile error: too many arguments
```

- In this case, there's no `put` function that accepts three arguments.

Argument Conversions

- The type of each parameter need not be exactly the same as the type of its corresponding argument.
  - C++ compilers apply some conversions in an attempt to find a match.
- The Standard C `getchar` returns the character read as an `int`, not a `char`.
  - `getchar` uses the integer value `EOF` to represent end-of-file and I/O failures.
- `putchar` has a parameter of type `int` (not `char`) to match `getchar`'s convention.
- So do these overloaded functions…

Argument Conversions

```
int put(int c);              // write c to standard output
int put(int c, FILE *f);     // write c to a designated file *f
```

- Even though they use `int` as a parameter type, you can pass them a `char` as the argument type:

```
char c;
~~~
put(c);                      // calls int put(int c);
put(c, stdout);              // calls int put(int c, FILE *f);
```

- In each case, the compiler promotes `char` to `int` to achieve the match.
  - That promotion may generate additional code that executes at run time.

15

Best Matches and Ambiguities

- In a group of overloaded functions, more than one function might be capable of satisfying a particular function call.

- For example, the first three of these functions are all capable of satisfying a call to `f(0)`:

```
int f(int i);                // f(0)?  maybe
long int f(long int li);     // f(0)?  maybe
char *f(char *p);            // f(0)?  maybe
int f(double d, int i);      // f(0)?  definitely not
```

- Clearly, the fourth function can't satisfy the call:
  - `f(0)` passes only one argument, while the fourth function requires two.

16

Best Matches and Ambiguities

- We're left with three viable functions:

```
int f(int i);
long int f(long int li);
char *f(char *p);
```

- The call `f(0)` can pass `0`:
  - as an `int`,
  - as a `long int`, or even
  - as a `char *` whose value is the null pointer.

17

Best Matches and Ambiguities

- When confronted with a choice of functions:
  - Overload resolution uses a ***ranking*** of the conversions from the argument type into the parameter type.
  - The ranking helps determine which function, if any, is the ***best match***.
- Thus, overload resolution depends on the exact type of each function call argument as well as the type of its corresponding parameter.
- In the case of calling `f(0)`, it depends on the exact type of argument `0`.
  - Obviously, `0` is an `int`.
  - But is it signed or unsigned?

18

Classifying Conversions

- The literal `0` is a plain `int`, which is `signed` by default.
- To find the best match for `f(0)`, consider the conversions…
- Calling `f(int)` requires no conversion at all.
  - It's an ***exact match***.
- Calling `f(long int)` requires converting `int` into "long int".
  - This is an ***integral conversion***.
- Calling `f(char *)` requires converting `int` into "pointer to char".
  - This is a ***pointer conversion***.

Exact Matches Always Win

- The compiler prefers "cheap" conversions.
- Exact matches are the cheapest of all.
- Thus, `f(0)` calls:

```
int f(int i);        // the winner for f(0)!
```

- It's the obvious choice.

Best Matches and Ambiguities

- Now, suppose we take away the exact match:

```
int f(int i);                 // f(int)
long int f(long int li);     // f(long)
char *f(char *pc);           // f(char *)
~~~
f(0);                         // best match?
```

21

Best Matches and Ambiguities

- This one might surprise you:

```
int f(int i);                 // f(int)
long int f(long int li);     // f(long)
char *f(char *pc);           // f(char *)
~~~
f(0);                         // compile error; why?
```

- C++ ranks all the standard conversions…

22

Standard Conversions by Rank

The choice:

`f(0)` ⇨ `f(long int)`
?

`f(0)` ⇨ `f(char *)`
?

Conversion	Rank
no conversion	Exact Match "cheap"
array-to-pointer conversion	
qualification conversion	
etc.	
integral promotion	Promotion
floating-point promotion	
integral conversion	Conversion "expensive"
floating-point conversion	
pointer conversion	
boolean conversion	
etc.	

23

Standard Conversions by Rank

`f(0)` ⇨ `f(long int)`

`f(0)` ⇨ `f(char *)`

Conversion	Rank
no conversion	Exact Match "cheap"
array-to-pointer conversion	
qualification conversion	
etc.	
integral promotion	Promotion
floating-point promotion	
integral conversion	Conversion "expensive"
floating-point conversion	
pointer conversion	
boolean conversion	
etc.	

24

Best Matches and Ambiguities

- In this case:
  - Calling `f(long int)` requires an ***integral conversion***.
  - Calling `f(char *)` requires a ***pointer conversion***.
- C++ considers these conversions to be equally "expensive".

```
long int f(long int li);
char *f(char *p);
~~~
f(0);           // compile error; a tie
```

- Thus, there's no unique best match.

Best Matches and Ambiguities

- When the compiler can't find at least one function that satisfies a particular call, it typically complains that there's ***"no matching function"***.
- When the compiler finds more than one function that satisfies the call, but no unique best match, it typically complains that the call is ***"ambiguous"***.

```
long int f(long int li);
char *f(char *p);
~~~
f();            // compile error: no matching function
f(0);           // compile error: ambiguous
```

Best Matches and Ambiguities

- The literal `0` is the only integer with an implicit conversion to a pointer type.
- The call `f(x)` is ambiguous only when `x` is the integer literal `0`:

```
long int f(long int li);
char *f(char *p);
int n;
~~~
f(n);           // calls long int f(long int)
f(1);           // calls long int f(long int)
```

27

Functions with Multiple Arguments

- How does overload resolution select the best match among functions with multiple arguments?
- For example, suppose that we have the following functions:

```
void f(double x, double y, double z);
void f(double x, int y, double z);
```

- If you eliminate either function, the following call with compile:

```
f(1.1, 2, 3);       // could work with either function by itself
```

- But which is the best match when they're overloaded?

28

Functions with Multiple Arguments

- Here's the general rule.
- Suppose we have a group of overloaded functions $\boldsymbol{F_1}$, $\boldsymbol{F_2}$, … $\boldsymbol{F_n}$.
- For a given call, function $\boldsymbol{F_i}$ is a better match than $\boldsymbol{F_j}$ if:
  - for every argument $\boldsymbol{A_k}$ in the call, $\boldsymbol{F_i}$'s conversion for $\boldsymbol{A_k}$ is no worse than $\boldsymbol{F_j}$'s conversion for $\boldsymbol{A_k}$, and
  - for at least one argument **A** in the call, $\boldsymbol{F_i}$'s conversion for $\boldsymbol{A}$ is better than $\boldsymbol{F_j}$'s conversion for $\boldsymbol{A}$.

29

Converting Multiple Arguments

- Here, again, is our set of overloaded functions:

```
void f(double x, double y, double z);
void f(double x, int y, double z);
```

- The best match for the following call is the 2nd function:

```
f(1.1, 2, 3);   // calls f(double, int, double)
```

- The conversions are the same for both functions on the 1st and 3rd arguments.
- The 2nd function is a better match on the 2nd (middle) argument.

30

Function Signatures

- Again, overloaded functions share the same name.
- Thus, it takes more than just a name to uniquely identify a particular C++ function.
- Thus, each function in a group of overloaded functions must have other properties that make it unique.
- The set of properties that uniquely identify a given function or function template is its ***signature***…

31

Function Signatures

- A function's ***signature*** is primarily the function's ***name*** and ***parameter type list***.
- The function's ***parameter type list*** is the sequence of types in the function's parameter list.
- For example, the declaration for one of standard `operator new` functions is:

```
void *operator new(std::size_t size, std::align_val_t alignment);
```

- Its parameter type list is:

```
(std::size_t, std::align_val_t)
```

32

Function Signatures

- The exact combination of additional properties that make up a function's signature depends on what "kind" of function it is.
- For example, a ***non-member function***'s signature also includes its enclosing namespace, if any.
- A class member function's signature also includes:
  - the class
  - *cv-qualifiers* (`const` and `volatile`), and
  - *ref-qualifiers* (`&` and `&&`).

Outline

- Function Overloading
- Name Lookup
- Default Function Arguments
- Function Templates
- Tying It All Together

Scope Regions and Name Lookup

- When the compiler encounters the declaration of a name, it stores that name and its attributes in a ***symbol table***.

- When the compiler encounters a reference to a name, it looks up the name in the symbol table to find those attributes.

```
class widget {              // stores widget
public:
    string name() const;    // looks up string; stores name
    ~~~
};

string s;                   // looks up string; stores s
widget w;                   // looks up widget; stores w
s = w.name();               // looks up s, w, and name
```

35

Name Lookup

- In C++, declarations can appear at:
  - ***local scope***: inside a function declaration, including that function's parameter list or a block nested inside a function definition
    - A name declared at local scope is in scope to the end of the function declaration or block containing that name.
  - ***class scope***: in the brace-enclosed body of a class definition
    - A name declared at class scope is in scope to the end of its class definition and within the parameter list and body of a member definition of the same class.
  - ***namespace scope***: outside of any function, class, structure, or union, *whether global or in some other namespace*
    - A name declared at *namespace* scope is in scope to the end of *its name-space definition, which for the global scope is the end of* its translation unit.

36

Qualified vs. Unqualified Names

- A name appearing just to the right of a `::`, `.` or `->` is a ***qualified name***.
- A name that's not qualified is an ***unqualified name***.
- For example:

```
string::assign      // string is unqualified
                    // assign is qualified
other.length()      // other  is unqualified
                    // length is qualified
words[i]->length()  // words  is unqualified
                    // i      is unqualified
                    // length is qualified
```

37

Qualified Name Lookup

- Lookup for qualified names is different from lookup for unqualified names.
- Lookup for qualified names is fairly simple:
  - For `S::n`, where `S` is a namespace:
    - Look for `n` in `S`.
  - For `T::n`, where `T` is a class type, or
  - for `x.n`, where `x` is a `T` object, or
  - for `p->n`, where `p` is a pointer to `T`:
    - Look for `n` in `T`.
    - If `n` isn't in `T`, look for `n` in `T`'s base classes (if any) from the direct base to the most indirect base.
- Failure to find the name is a compile error.

38

Unqualified Name Lookup

- Name lookup for unqualified names in C++ is an extension of name lookup in C.
- In this example, `m` and `n` are unqualified names appearing in a non-member function at namespace scope:

```
namespace S {
    int m, n;
    void f(int n) {
        m = n;          // m refers to S::m; n refers to parameter
    }
}
```


Unqualified Name Lookup

- For an unqualified name `n` appearing in a non-member function `f` at namespace scope:
  - ***Look in the local scope(s).***
    - That is, look for `n` in the scope of the block in which `n` appears.
    - Work outward to the scope of `f`.
  - ***Look in the namespace scope(s).***
    - That is, look for `n` in the namespace scope(s) enclosing `f`.
    - Start in the namespace immediately enclosing `f`.
    - Work outward to the global scope.
  - ***Stop as soon as you find any declaration for n.***
- Again, failure to find the name is a compile error.

Unqualified Name Lookup

- In this example, `m` and `n` are unqualified names appearing in function `f` that's a member of a class `T` within a namespace `S`:

```
namespace S {
    int m, n;
    class T {
        void f(int n);
    };
    void T::f(int n) {
        m = n;          // m refers to S::m; n refers to parameter
    }
}
```

41

Unqualified Name Lookup

- For an unqualified name `n` appearing in a function `f` that's a member of class `T`:
  - ***Look in f's local scope(s).***
  - ***Look in the class scope(s).***
    - That is, look for `n` in the scope of `T`.
    - If `n` isn't in `T`, look for `n` in `T`'s base classes (if any) from the direct base to the most indirect base.
    - If the compiler finds `n` as a class member, it interprets `n` as `this->n`.
  - ***Look in the namespace scope(s).***
  - ***Again, stop as soon as you find any declaration for n.***
- As always, failure to find the name is a compile error.

42

Name Lookup Precedes Overload Resolution

- Name lookup and overload resolution are distinct compilation steps.
- When a name, say ***f***, occurs as a function name in a call, name lookup seeks the first scope that declares at least one instance of ***f***.
- If name lookup succeeds — it finds a scope containing at least one ***f*** — then all the ***f***'s in that scope become the ***candidates*** for the call…

43

Name Lookup Precedes Overload Resolution

- After name lookup, overload resolution tries to identify the best matching function among the candidates.
- Overload resolution must find a unique best match.
- Otherwise, the call produces a compile error.
- If there's a better matching function in an outer scope…
  - name lookup won't find it, and
  - overload resolution won't consider it.

44

Name Lookup Precedes Overload Resolution

- For example, the call on (5) fails because name lookup finds only the put declarations at (3) and (4), not the declarations at (1) and (2):

```
int put(int c);                        // (1) not found
int put(int c, FILE *f);               // (2) not found
namespace N {
    int put(char const *s);            // (3) found
    int put(char const *s, FILE *f);   // (4) found
    void f(int c) {
        put(c);                        // (5) error: no valid match
        ~~~
    }
}
```

- Neither the declaration at (3) nor (4) is a viable match, so the call fails.

Unqualified Name Lookup and Inheritance

- Similarly, Derived::f hides Base::f rather than overloading with it:

```
class Base {
public:
    void f(int n);
};

class Derived: public Base {
public:
    void f(double d);       // hides Base::f(int)
};

Derived dx;
dx.f(3);                    // calls Derived::f(double)
```

Argument-Dependent Lookup

- Actually, it's not *quite* true that overloaded functions "must be in the same scope".
- There's one more facet of unqualified name lookup to consider: ***argument-dependent lookup (ADL)***.
- ADL is specifically for unqualified function names in function calls.
- ADL adds this name lookup rule:
  - For each argument in the function call whose type is declared in a name-space, look in that namespace for the function name, as well as in other scopes searched by the usual name lookup.

47

Argument-Dependent Lookup

- Here's an example of what would happen if ADL weren't a part of C++:

```
namespace N {
    class T;
    void f(T &r);
}

N::T x;
~~~
f(x);       // compile error w/o ADL: never looks in N
N::f(x);    // OK w/o ADL: finds f in N
```

48

Argument-Dependent Lookup

- At first glance, this may not look so bad — the programmer simply needs to specify that `f` comes from namespace `N`.

```
N::f(x);
```

- However, suppose that:
  - namespace `N` is actually `std`,
  - class `T` is actually `string`, and
  - function `f` is actually `operator<<`.
- Altogether, those changes yield this simplified version of code from the standard *<string>* header…

Argument-Dependent Lookup

```
// <string>

#include <iosfwd>

namespace std {
    class string;
    ostream &operator<<(ostream &, string const &);
    ~~~
}
```

- Again, this is a simplified version of the *<string>* header.
- The following code uses this header…

Argument-Dependent Lookup

- Without ADL, the compiler would reject these calls to `operator<<` because it wouldn't look for `operator<<` in namespace `std`:

```
#include <iostream>
#include <string>

std::string s;
std::cout << s;                // compile error w/o ADL
operator<<(std::cout, s);      // compile error w/o ADL
```

- You'd need to call it using the function syntax and explicitly specify that `operator<<` is a member of `std`:

```
std::operator<<(std::cout, s);  // OK w/o ADL
```


Sutter's Interface Principle

- Sutter [2000] offers some advice for grouping code into namespaces based on what he calls the ***Interface Principle***:
  - For a class `X`, all functions, including free functions, that both "mention" `X` and are "supplied with" `X` are logically part of `X`, because they form part of the interface of `X`.

- For example, this function "mentions" `string` and is "supplied with" `string`:

```
ostream &operator<<(ostream &, string const &);
```

- Thus, it's part of `string`'s interface, even though it's not a member, and may not even be a friend.

Sutter's Interface Principle

- He then offers this advice:

✓ *If you put a class into a namespace, be sure to put of its interface functions into the same namespace.*

- If you don't, you may be unhappy with how your code behaves.

ADL and Overloading

- In a function call, overload resolution also considers function declarations found by ADL.
- That is, function declarations found by ADL are added to the candidate set as if they had been found through unqualified name lookup.

Outline

- Function Overloading
- Name Lookup
- Default Function Arguments
- Function Templates
- Tying It All Together

Default Arguments

- In the Standard Library, `stdout` is an object that can be passed as a `FILE *` to represent the standard output stream.

- Here is a pair of overloaded functions we saw earlier:

```
int put(int c);             // write c to standard output
int put(int c, FILE *f);    // write c to a designated file *f
```

- You could declare it as a single function with a ***default argument*** for the second parameter:

```
int put(int c, FILE *f = stdout);
```

- With this change, the compiler produces slightly different code…

How Many Functions?

- When declared as a pair of overloaded function, each function has a distinct signature:
  - `int put(int c)`
    - is a function named "put" whose parameter type list is `(int)`.
  - `int put(int c, FILE *f)`
    - is a function named "put" whose parameter type list is `(int, FILE *)`.
- When declared as a single function with a default argument, it has the same signature as the second overloaded function:
  - `int put(int c, FILE *f = stdout)`
    - is a function named "put" whose parameter type list is `(int, FILE *)`.
- The default argument value is *not* part of the signature.

Default Arguments and Overload Resolution

- When measuring the "cost" of selecting an overloaded function, filling in a default argument is considered "free".
- For example, suppose that we have the following two overloaded functions:

```
void g(double d);
void g(int x, int y = 1);
```

- Which is the best match for calling `g(0)`?
- Consider the conversions…

Default Arguments and Overload Resolution

- Calling `g(double)` requires converting `int` into `double`.
  - This is a ***floating-point conversion***.
- Calling `g(int, int = 1)` requires no conversion at all.
  - `0` is an ***exact match*** for the first parameter `int`.
  - Applying the default argument to the second `int` parameter isn't considered a "conversion" of any kind.
- Thus, `g(0)` calls:

```
void g(int x, int y = 1);       // the winner for g(0)!
```

59

Outline

- Function Overloading
- Name Lookup
- Default Function Arguments
- **Function Templates**
- Tying It All Together

60

Function Templates

- A ***function template*** is a generalization of an algorithm.
  - It's ***not an actual function***.
- Rather, it's a single declaration that can generate declarations for similar, but distinct functions.
- Each generated function implements the algorithm for operands of different types.

Function Templates

- For example, the standard <*algorithm*> header provides a `max` function template that looks something like this:

```
template <typename T>
constexpr T const &max(T const &a, T const &b) {
    return (a > b) ? a : b;
}
```

Template Argument Deduction

- C++ often lets you omit the angle-bracketed template argument list from a call to a function template.
- That is, the call can use just the template name as the function name.

```
int x = 10;
int y = 20;
int z = max(x, y);  // max, not max<int>
```

- In this case, the compiler performs ***template argument deduction***:
  - It deduces the template argument(s) from the function call argument(s).

63

Template Argument Deduction

- Template argument deduction makes a function template look more like an unbounded set of overloaded functions, as in:

```
template <typename T>
constexpr T const &max(T const &a, T const &b);
```

- C++ can deduce the type argument for each of these calls:

```
int i, j;
float f, g;
~~~
int k = max(i, j);      // calls max<int>(i, j)
float h = max(f, g);    // calls max<float>(f, g)
```

64

Specified and Deduced Type Arguments

- When you call a function template, you can either specify the template arguments explicitly or let the compiler deduce them:

```
int i, j;
float f, g;
~~~
int k = max<int>(i, j);     // (1) type specified as int
float h = max(f, g);        // (2) type deduced as float
```

- However, these calls produce slightly different behavior…

Specified and Deduced Type Arguments

- When performing template argument deduction, the compiler doesn't consider most type conversions.
- For example, although there's normally a standard conversion from `int` to `double`, the compiler won't convert `x` into a `double` in this call:

```
int x = 1;
double y = 2.5;
double z = max(x, y);   // compile error: can't deduce T
```

- Instead, the compiler rejects the call.

Specified and Deduced Type Arguments

- However, if you explicitly call `max<double>`, the compiler will perform the implicit conversion:

```
int x = 1;
double y = 2.5;
double z = max<double>(x, y);   // OK: converts x to double
```

- Similarly, calling `max<int>` would implicitly convert `y` into an `int`:

```
int n = max<int>(x, y);         // OK: converts y to int
```


Overloading and Templates

- You can overload functions and function templates with each other.
- When an overloaded function and function template provide equally good matches, the compiler chooses the non-template function as the better match.
- Using this behavior, you can effectively customize a function template for specific argument types…

Overloading and Templates

- For example, you could write a `max` function for C-strings that uses `strcmp` to go along with the general `max` template:

```
template <typename T>
constexpr T const &max(T const &a, T const &b);

constexpr char const *max(char const *a, char const *b) {
    return strcmp(a, b) > 0 ? a : b;
}
```


Overloading and Templates

- This call invokes the `max` that specifically takes `char const *` objects:

```
char const N[] = "Nancy";
char const D[] = "Dan";
char const *p;
~~~
p = max(D, N);  // calls non-template max
```

- In the very-unlikely event that you actually want to call the template version of `max` on `char const *` objects, you can do so explicitly:

```
p = max<char const *>(D, N);
```

Outline

- Function Overloading
- Name Lookup
- Default Function Arguments
- Function Templates
- Tying It All Together

The `std::swap` Two-Step

- The most common example that combines most of these features is probably the "`std::swap` Two-Step" [Niebler 2014].
- The "Two-Step" is a programming technique that allows for a ***customization point***.
  - It's especially valuable for library authors writing templates that will be widely used, though it also has other uses.
- In the titular example, that customization point is a `swap` function…

The std::swap Two-Step

- Here's a possible implementation for the `std::swap` function template from *<algorithm>*:

```
template <typename T>
void std::swap(T &a, T &b) {
    T temp {std::move(a)};
    a = std::move(b);
    b = std::move(temp);
}
```

- Because this function template uses basic move semantics, it accepts a wide variety of types.

- However, a type-specific `swap` function can be more efficient…

The std::swap Two-Step

- For example, the `Saks::string` class from earlier might benefit from providing its own "custom" `swap` function:

```
namespace Saks {
    class string {
        ~~~
        friend void swap(string &a, string &b);
    private:
        std::size_t stored_length;
        char *actual_str;
    };
    void swap(string &a, string &b);
}
```

- You'll see why I've written `swap` as a friend function shortly.

The std::swap Two-Step

- This swap function is probably more efficient for Saks::strings, even taking move semantics into account:

```
void Saks::swap(string &a, string &b) {
    std::swap(a.stored_length, b.stored_length);
    std::swap(a.actual_str, b.actual_str);
}
```

- Unfortunately, it's still possible to use std::swap with Saks::strings by accident…

75

The std::swap Two-Step

- For example, here's another type Person with a custom swap function that has a Saks::string as a data member:

```
class Person {
public:
    ~~~
    void swap(Person &other);
    ~~~
private:
    Saks::string name;
    unsigned idnum;
};
```

76

The std::swap Two-Step

- If you didn't know that Saks::string had a custom swap function, you might write Person::swap like this:

```
void Person::swap(Person &other) {
    std::swap(name, other.name);    // oops, uses std::swap
    std::swap(idnum, other.idnum);
};
```

- Fortunately, because of how Saks::string provides its custom swap function, there's a simple technique that you can use to avoid this mistake…

The std::swap Two-Step

- If you instead write Person::swap like this, the compiler will find the custom swap function for Saks::string:

```
void Person::swap(Person &other) {
    using std::swap;
    swap(name, other.name);     // OK, uses Saks::swap
    swap(idnum, other.idnum);
};
```

- If Saks::string doesn't have a custom swap function, the compiler will automatically fall back to using std::swap.
- Although you can apply this technique easily, the effect is fairly complex…

How It Works

- If you call the function as `std::swap`, the compiler uses qualified name lookup and goes directly to namespace `std` to find the `swap` function:

```
void Person::swap(Person &other) {
    std::swap(name, other.name);
    std::swap(idnum, other.idnum);
};
```

79

How It Works

- However, if you call the function as simply `swap`, the compiler searches for `swap` using *unqualified* name lookup:

```
void Person::swap(Person &other) {
    using std::swap;
    swap(name, other.name);
    swap(idnum, other.idnum);
};
```

- In other words, the compiler:
  - searches outward for declarations for `swap` from the point of the call, and
  - allows argument-dependent lookup (ADL).

80

How It Works — Name Lookup

- Because of the local using-declaration, the compiler immediately finds `std::swap` as if it were declared locally:

```
void Person::swap(Person &other) {
    using std::swap;
    swap(name, other.name);
    swap(idnum, other.idnum);
};
```

- Thus, the `std::swap` function template is a candidate for the calls to `swap`.
- However, the compiler also performs ADL…

How It Works — Name Lookup

- Because `name` is a `Saks::string` object, the compiler also finds the `swap` in namespace `Saks`:

```
namespace Saks {
    class string { ~~~ };
    void swap(string &a, string &b);
}
```

- Thus, `Saks::swap` is also a candidate for the calls to `swap`.
- Now, the compiler performs overload resolution to choose between `std::swap` and `Saks::swap`…

How It Works — Overload Resolution

- Both `std::swap<Saks::string>` and `Saks::swap` take two arguments of type `string &`.
- Thus, both functions provide exact matches for both arguments.
- However, `std::swap` is a function template, while `Saks::swap` is an ordinary function.
- Thus, the compiler selects `Saks::swap` as the best match.
- If `Saks::swap` hadn't existed, `std::swap` would've been the only candidate function, and thus would've been the best match by default.

Important Details

- Note that this technique works specifically because both `std::swap` and `Saks::swap` are non-member functions that take two arguments.
- If `Saks::string`'s `swap` function had been a member function, you would've needed to write the calls differently:

```
void Person::swap(Person &other) {
    name.swap(other.name);
    std::swap(idnum, other.idnum);
};
```

- That being the case, `Person::swap` should probably be rewritten as a non-member function as well.

Important Details

- It's also important that `Saks::string`'s `swap` function is part of the `Saks` namespace.
- If the custom `swap` function was somewhere else, ADL wouldn't find it.
- Remember: the local using-declaration for `std::swap` ends the normal name lookup process immediately.
  - There's nowhere else that you could put the custom `swap` function where it would be selected as a candidate.

85

Takeaways

- Name lookup precedes overload resolution.
- To allow for ADL, remember Sutter's advice regarding interface design:
  - If you put a class into a namespace, be sure to put all helper functions and operators into the same namespace.
- When a function and a function template are equally good matches for a call, overload resolution favors the non-template.
- Be aware of customization points, and use the Two-Step technique to avoid accidentally calling the non-custom version of a function.

86

Bibliography

- Niebler [2014]. *Customization Point Design in C++11 and Beyond*. https://ericniebler.com/2014/10/21/customization-point-design-in-c11-and-beyond/
- Sutter [2000]. Herb Sutter, *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley.