11.3 — Function overload resolution and ambiguous matches

ALEX DECEMBER 28, 2023

In the previous lesson (11.2 -- Function overload differentiation (https://www.learncpp.com/cpp-tutorial/function-overload-differentiation/)), we discussed which attributes of a function are used to differentiate overloaded functions from each other. If an overloaded function is not properly differentiated from the other overloads of the same name, then the compiler will issue a compile error.

However, having a set of differentiated overloaded functions is only half of the picture. When any function call is made, the compiler must also ensure that a matching function declaration can be found.

With non-overloaded functions (functions with unique names), there is only one function that can potentially match a function call. That function either matches (or can be made to match after type conversions are applied), or it doesn't (and a compile error results). With overloaded functions, there can be many functions that can potentially match a function call. Since a function call can only resolve to one of them, the compiler has to determine which overloaded function is the best match. The process of matching function calls to a specific overloaded function is called **overload resolution**.

In simple cases where the type of the function arguments and type of the function parameters match exactly, this is (usually) straightforward:

```
#include <iostream>

void print(int x)
{
    std::cout << x << '\n';
}

void print(double d)
{
    std::cout << d << '\n';
}

int main()
{
    print(5); // 5 is an int, so this matches print(int)
    print(6.7); // 6.7 is a double, so this matches print(double)

    return 0;
}
```

But what happens in cases where the argument types in the function call don't exactly match the parameter types in any of the overloaded functions? For example:

```
#include <iostream>

void print(int x)
{
	std::cout << x << '\n';
}

void print(double d)
{
	std::cout << d << '\n';
}

int main()
{
	print('a'); // char does not match int or double
	print(5L); // long does not match int or double

	return 0;
}
```

Just because there is no exact match here doesn't mean a match can't be found -- after all, a `char` or `long` can be implicitly type converted to an `int` or a `double`. But which is the best conversion to make in each case?

In this lesson, we'll explore how the compiler matches a given function call to a specific overloaded function.

Resolving overloaded function calls

When a function call is made to an overloaded function, the compiler steps through a sequence of rules to determine which (if any) of the overloaded functions is the best match.

At each step, the compiler applies a bunch of different type conversions to the argument(s) in the function call. For each conversion applied, the compiler checks if any of the overloaded functions are now a match. After all the different type conversions have been applied and checked for matches, the step is done. The result will be one of three possible outcomes:

- No matching functions were found. The compiler moves to the next step in the sequence.
- A single matching function was found. This function is considered to be the best match. The matching process is now complete, and subsequent steps are not executed.
- More than one matching function was found. The compiler will issue an ambiguous match compile error. We'll discuss this case further in a bit.

If the compiler reaches the end of the entire sequence without finding a match, it will generate a compile error that no matching overloaded function could be found for the function call.

The argument matching sequence

Step 1) The compiler tries to find an exact match. This happens in two phases. First, the compiler will see if there is an overloaded function where the type of the arguments in the function call exactly matches the type of the parameters in the overloaded functions. For example:

```
void print(int)
{
}

void print(double)
{
}

int main()
{
    print(0); // exact match with print(int)
    print(3.4); // exact match with print(double)

    return 0;
}
```

Because the `0` in the function call `print(0)` is an int, the compiler will look to see if a `print(int)` overload has been declared. Since it has, the compiler determines that `print(int)` is an exact match.

Second, the compiler will apply a number of trivial conversions to the arguments in the function call. The **trivial conversions** are a set of specific conversion rules that will modify types (without modifying the value) for purposes of finding a match. For example, a non-const type can be trivially converted to a const type:

```
void print(const int)
{
}

void print(double)
{
}

int main()
{
    int x { 0 };
    print(x); // x trivially converted to const int

    return 0;
}
```

In the above example, we've called `print(x)`, where `x` is an `int`. The compiler will trivially convert `x` from an `int` into a `const int`, which then matches `print(const int)`.

For advanced readers

Converting a non-reference type to a reference type (or vice-versa) is also a trivial conversion.

Matches made via the trivial conversions are considered exact matches.

Step 2) If no exact match is found, the compiler tries to find a match by applying numeric promotion to the argument(s). In lesson (10.1 -- Implicit type conversion (https://www.learncpp.com/cpp-tutorial/implicit-type-conversion/)), we covered how certain narrow integral and floating point types can be automatically promoted to wider types, such as `int` or `double`. If, after numeric promotion, a match is found, the function call is resolved.

For example:

```
void print(int)
{
}

void print(double)
{
}

int main()
{
    print('a'); // promoted to match print(int)
    print(true); // promoted to match print(int)
    print(4.5f); // promoted to match print(double)

    return 0;
}
```

For `print('a')`, because an exact match for `print(char)` could not be found in the prior step, the compiler promotes the char `'a'` to an `int`, and looks for a match. This matches `print(int)`, so the function call resolves to `print(int)`.

Step 3) If no match is found via numeric promotion, the compiler tries to find a match by applying numeric conversions (10.3 -- Numeric conversions (https://www.learncpp.com/cpp-tutorial/numeric-conversions/)) to the arguments.

For example:

```
#include <string> // for std::string

void print(double)
{
}

void print(std::string)
{
}

int main()
{
    print('a'); // 'a' converted to match print(double)

    return 0;
}
```

In this case, because there is no `print(char)` (exact match), and no `print(int)` (promotion match), the `'a'` is numerically converted to a double and matched with `print(double)`.

Key insight

Matches made by applying numeric promotions take precedence over any matches made by applying numeric conversions.

Step 4) If no match is found via numeric conversion, the compiler tries to find a match through any user-defined conversions. Although we haven't covered user-defined conversions yet, certain types (e.g. classes) can define conversions to other types that can be implicitly invoked. Here's an example, just to illustrate the point:

```
// We haven't covered classes yet, so don't worry if this doesn't make sense
class X // this defines a new type called X
{
public:
    operator int() { return 0; } // Here's a user-defined conversion from X to int
};

void print(int)
{
}

void print(double)
{
}

int main()
{
    X x; // Here, we're creating an object of type X (named x)
    print(x); // x is converted to type int using the user-defined conversion from X to int

    return 0;
}
```

In this example, the compiler will first check whether an exact match to `print(X)` exists. We haven't defined one. Next the compiler will check whether `x` can be numerically promoted, which it can't. The compiler will then check if `x` can be numerically converted, which it also can't. Finally, the compiler will then look for any user-defined conversions. Because we've defined a user-defined conversion from `X` to `int`, the compiler will convert `X` to an `int` to match `print(int)`.

After applying a user-defined conversion, the compiler may apply additional implicit promotions or conversions to find a match. So if our user-defined conversion had been to type `char` instead of `int`, the compiler would have used the user-defined conversion to `char` and then promoted the result into an `int` to match.

Related content

We discuss how to create user-defined conversions for class types (by overloading the typecast operators) in lesson 21.11 -- Overloading typecasts (https://www.learncpp.com/cpp-tutorial/overloading-typecasts/).

For advanced readers

The constructor of a class also acts as a user-defined conversion from other types to that class type, and can be used during this step to find matching functions.

Step 5) If no match is found via user-defined conversion, the compiler will look for a matching function that uses ellipsis.

Related content

We cover ellipses in lesson 20.5 -- Ellipsis (and why to avoid them) (https://www.learncpp.com/cpp-tutorial/ellipsis-and-why-to-avoid-them/).

Step 6) If no matches have been found by this point, the compiler gives up and will issue a compile error about not being able to find a matching function.

Ambiguous matches

With non-overloaded functions, each function call will either resolve to a function, or no match will be found and the compiler will issue a compile error:

```
void foo()
{
}

int main()
{
     foo(); // okay: match found
     goo(); // compile error: no match found

     return 0;
}
```

With overloaded functions, there is a third possible outcome: an `ambiguous match` may be found. An **ambiguous match** occurs when the compiler finds two or more functions that can be made to match in the same step. When this occurs, the compiler will stop matching and issue a compile error stating that it has found an ambiguous function call.

Since every overloaded function must be differentiated in order to compile, you might be wondering how it is possible that a function call could result in more than one match. Let's take a look at an example that illustrates this:

```
void print(int)
{
}

void print(double)
{
}

int main()
{
    print(5L); // 5L is type long

    return 0;
}
```

Since literal `5L` is of type `long`, the compiler will first look to see if it can find an exact match for `print(long)`, but it will not find one. Next, the compiler will try numeric promotion, but values of type `long` can't be promoted, so there is no match here either.

Following that, the compiler will try to find a match by applying numeric conversions to the `long` argument. In the process of checking all the numeric conversion rules, the compiler will find two potential matches. If the `long` argument is numerically converted into an `int`, then the function call will match `print(int)`. If the `long` argument is instead converted into a `double`, then it will match `print(double)` instead. Since two possible matches via numeric conversion have been found, the function call is considered ambiguous.

On Visual Studio 2019, this results in the following error message:

```
error C2668: 'print': ambiguous call to overloaded function
message : could be 'void print(double)'
message : or       'void print(int)'
message : while trying to match the argument list '(long)'
```

Key insight

If the compiler finds multiple matches in a given step, an ambiguous function call will result. This means no match from a given step is considered to be better than any other match from the same step.

Here's another example that yields ambiguous matches:

```
void print(unsigned int)
{
}

void print(float)
{
}

int main()
{
    print(0); // int can be numerically converted to unsigned int or to float
    print(3.14159); // double can be numerically converted to unsigned int or to float

    return 0;
}
```

Although you might expect `0` to resolve to `print(unsigned int)` and `3.14159` to resolve to `print(float)`, both of these calls result in an ambiguous match. The `int` value `0` can be numerically converted to either an `unsigned int` or a `float`, so either overload matches equally well, and the result is an ambiguous function call.

The same applies for the conversion of a `double` to either a `float` or `unsigned int`. Both are numeric conversions, so either overload matches equally well, and the result is again ambiguous.

Resolving ambiguous matches

Because ambiguous matches are a compile-time error, an ambiguous match needs to be disambiguated before your program will compile. There are a few ways to resolve ambiguous matches:

1. Often, the best way is simply to define a new overloaded function that takes parameters of exactly the type you are trying to call the function with. Then C++ will be able to find an exact match for the function call.
2. Alternatively, explicitly cast the ambiguous argument(s) to match the type of the function you want to call. For example, to have `print(0)` match `print(unsigned int)` in the above example, you would do this:

```
int x{ 0 };
print(static_cast<unsigned int>(x)); // will call print(unsigned int)
```

3. If your argument is a literal, you can use the literal suffix to ensure your literal is interpreted as the correct type:

```
print(0u); // will call print(unsigned int) since 'u' suffix is unsigned int, so this is now an exact match
```

The list of the most used suffixes can be found in lesson 5.2 -- Literals (https://www.learncpp.com/cpp-tutorial/literals/).

Matching for functions with multiple arguments

If there are multiple arguments, the compiler applies the matching rules to each argument in turn. The function chosen is the one for which each argument matches at least as well as all the other functions, with at least one argument matching better than all the other functions. In other words, the function chosen must provide a better match than all the other candidate functions for at least one parameter, and no worse for all of the other parameters.

In the case that such a function is found, it is clearly and unambiguously the best choice. If no such function can be found, the call will be considered ambiguous (or a non-match).

For example:

```
#include <iostream>

void print(char, int)
{
    std::cout << 'a' << '\n';
}

void print(char, double)
{
    std::cout << 'b' << '\n';
}

void print(char, float)
{
    std::cout << 'c' << '\n';
}

int main()
{
    print('x', 'a');

    return 0;
}
```

In the above program, all functions match the first argument exactly. However, the top function matches the second parameter via promotion, whereas the other functions require a conversion. Therefore, `print(char, int)` is unambiguously the best match.

Next lesson
11.4 Deleting functions

Back to table of contents

Previous lesson
11.2 Function overload differentiation