7.6 — Function overloading

BY ALEX ON AUGUST 3RD, 2007 | LAST MODIFIED BY ALEX ON JUNE 29TH, 2018

Function overloading is a feature of C++ that allows us to create multiple functions with the same name, so long as they have different parameters. Consider the following function:

```
int add(int x, int y)
{
    return x + y;
}
```

This trivial function adds two integers. However, what if we also need to add two floating point numbers? This function is not at all suitable, as any floating point parameters would be converted to integers, causing the floating point arguments to lose their fractional values.

One way to work around this issue is to define multiple functions with slightly different names:

```
int addInteger(int x, int y)
{
    return x + y;
}

double addDouble(double x, double y)
{
    return x + y;
}
```

However, for best effect, this requires that you define a consistent naming standard, remember the name of all the different flavors of the function, and call the correct one.

Function overloading provides a better solution. Using function overloading, we can simply declare another add() function that takes double parameters:

```
double add(double x, double y)
{
    return x + y;
}
```

We now have two version of add():

```
int add(int x, int y); // integer version
double add(double x, double y); // floating point version
```

Although you might expect this to cause a naming conflict, that is not the case here. The compiler is able to determine which version of add() to call based on the arguments used in the function call. If we provide two ints, C++ will know we mean to call add(int, int). If we provide two floating point numbers, C++ will know we mean to call add(double, double). In fact, we can define as many overloaded add() functions as we want, so long as each add() function has unique parameters.

Consequently, it's also possible to define add() functions with a differing number of parameters:

```
int add(int x, int y, int z)
{
    return x + y + z;
}
```

Even though this add() function has 3 parameters instead of 2, because the parameters are different than any other version of add(), this is valid.

Function return types are not considered for uniqueness

A function's return type is NOT considered when overloading functions. (Note for advanced readers: This was an intentional choice, as it ensures the behavior of a function call or subexpression can be determined independently from the rest of the expression, making understanding complex expressions much simpler. Put another way, we can always determine which version of a function will be

called based solely on the arguments. If return values were included, then we wouldn't have an easy syntactic way to tell which version of a function was being called -- we'd also have to understand how the return value was being used, which requires a lot more analysis).

Consider the case where you want to write a function that returns a random number, but you need a version that will return an int, and another version that will return a double. You might be tempted to do this:

```
int getRandomValue();
double getRandomValue();
```

The compiler will flag this as an error. These two functions have the same parameters (none), and consequently, the second getRandomValue() will be treated as an erroneous redeclaration of the first.

The best way to address this is to give the functions different names:

```
int getRandomInt();
double getRandomDouble();
```

An alternative method is to make the functions return void, and have the return value passed back to the caller as an out parameter (see lesson **7.3 -- Passing arguments by reference** if you need a reminder what an out parameter is).

```
void getRandomValue(int &out);
void getRandomValue(double &out);
```

Because these functions have different parameters, they are considered unique. However, there are downsides to doing this. First, the syntax is awkward, and you can't route the output of this function directly into the input of another. Consider:

```
// method 1: getRandomInt() returns an int
printValue(getRandomInt()); // easy

// method 2: getRandomValue() has an int out parameter
int temp; // now we need a temp variable
getRandomValue(temp); // to call getRandomValue(int) here
printValue(temp); // this has to be a separate line since getRandomValue() returns void
```

Also, the type of the argument passed in must match the type of the parameter exactly. For these reasons, we don't recommend this method.

Typedefs are not distinct

Since declaring a typedef does not introduce a new type, the following two declarations of Print() are considered identical:

```
typedef char *string;
void print(string value);
void print(char *value);
```

How function calls are matched with overloaded functions

Making a call to an overloaded function results in one of three possible outcomes:

1) A match is found. The call is resolved to a particular overloaded function.
2) No match is found. The arguments can not be matched to any overloaded function.
3) An ambiguous match is found. The arguments matched more than one overloaded function.

When an overloaded function is called, C++ goes through the following process to determine which version of the function will be called:

1) First, C++ tries to find an exact match. This is the case where the actual argument exactly matches the parameter type of one of the overloaded functions. For example:

```
void print(char *value);
void print(int value);

print(0); // exact match with print(int)
```

Although 0 could technically match print(char*) (as a null pointer), it exactly matches print(int) (matching char* would require an implicit conversion). Thus print(int) is the best match available.

2) If no exact match is found, C++ tries to find a match through promotion. In lesson **4.4 -- type conversion and casting**, we covered how certain types can be automatically promoted via internal type conversion to other types. To summarize,

- Char, unsigned char, and short is promoted to an int.
- Unsigned short can be promoted to int or unsigned int, depending on the size of an int
- Float is promoted to double
- Enum is promoted to int

For example:

```
void print(char *value);
void print(int value);

print('a'); // promoted to match print(int)
```

In this case, because there is no print(char), the char 'a' is promoted to an integer, which then matches print(int).

3) If no promotion is possible, C++ tries to find a match through standard conversion. Standard conversions include:

- Any numeric type will match any other numeric type, including unsigned (e.g. int to float)
- Enum will match the formal type of a numeric type (e.g. enum to float)
- Zero will match a pointer type and numeric type (e.g. 0 to char*, or 0 to float)
- A pointer will match a void pointer

For example:

```
struct Employee; // defined somewhere else
void print(float value);
void print(Employee value);

print('a'); // 'a' converted to match print(float)
```

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

Note that all standard conversions are considered equal. No standard conversion is considered better than any of the others.

4) Finally, C++ tries to find a match through user-defined conversion. Although we have not covered classes yet, classes (which are similar to structs) can define conversions to other types that can be implicitly applied to objects of that class. For example, we might define a class X and a user-defined conversion to int.

```
class X; // with user-defined conversion to int

void print(float value);
void print(int value);

X value; // declare a variable named value of type class X
print(value); // value will be converted to an int and matched to print(int)
```

Although value is of type class X, because this particular class has a user-defined conversion to int, the function call print(value) will resolve to the Print(int) version of the function.

We will cover the details on how to do user-defined conversions of classes when we cover classes.

Ambiguous matches

If every overloaded function has to have unique parameters, how is it possible that a call could result in more than one match? Because all standard conversions are considered equal, and all user-defined conversions are considered equal, if a function call matches multiple candidates via standard conversion or user-defined conversion, an ambiguous match will result. For example:

```
void print(unsigned int value);
void print(float value);

print('a');
print(0);
print(3.14159);
```

In the case of `print('a')`, C++ can not find an exact match. It tries promoting 'a' to an int, but there is no print(int) either. Using a standard conversion, it can convert 'a' to both an unsigned int and a floating point value. Because all standard conversions are considered equal, this is an ambiguous match.

`print(0)` is similar. 0 is an int, and there is no print(int). It matches both calls via standard conversion.

`print(3.14159)` might be a little surprising, as most programmers would assume it matches print(float). But remember that all literal floating point values are doubles unless they have the 'f' suffix. 3.14159 is a double, and there is no print(double). Consequently, it matches both calls via standard conversion.

Ambiguous matches are considered a compile-time error. Consequently, 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 the type of the function you want to call. For example, to have print(0) call the print(unsigned int), 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
```

The list of the most used suffixes can be found in lesson **2.8 -- Literals**.

Matching for functions with multiple arguments

If there are multiple arguments, C++ 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 fcn(char c, int x)
{
    std::cout << 'a';
}

void fcn(char c, double x)
{
    std::cout << 'b';
}

void fcn(char c, float x)
{
    std::cout << 'c';
}

int main()
{
    fcn('x', 4);
}
```

In the above program, all functions match the first argument exactly. However, the top function matches the second parameter exactly, whereas the other functions require a conversion. Therefore, the top function (the one that prints 'a') is unambiguously the best match.

Conclusion

Function overloading can lower a program's complexity significantly while introducing very little additional risk. Although this particular lesson is long and may seem somewhat complex (particularly the matching rules), in reality function overloading typically works transparently and without any issues. The compiler will flag all ambiguous cases, and they can generally be easily resolved.

Rule: use function overloading to make your program simpler.


7.7 -- Default parameters


Index


7.5 -- Inline functions

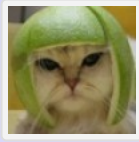
Share this:

C++ TUTORIAL | PRINT THIS POST

78 comments to 7.6 — Function overloading

Ramesh
June 19, 2018 at 7:10 am · Reply

I am not able to understand this section-> Matching for functions with multiple arguments
Could you give some example programs?

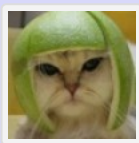


Alex
June 29, 2018 at 12:40 pm · Reply

I added an example. Does that help?

Peter Baum
May 9, 2018 at 2:56 pm · Reply

Regarding: "Note that the function's return type is NOT considered when overloading functions."
Here is an example of what looks like an arbitrary rule, and therefore yet another arbitrary rule for the student to memorize. Perhaps it would be better to explain that the compiler would have a hard time choosing the required function from the return type (at least in many instances). With such an explanation, it isn't just an arbitrary rule, but something that actually makes sense.



Alex
May 10, 2018 at 10:30 pm · Reply

Good point. I added a little context about why this choice was intentionally made, for readers who care. Is the explanation clear or confusing?

Peter Baum