

7.6 — Function overloading

BY ALEX ON AUGUST 3RD, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

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:

```
1 int add(int x, int y)
2 {
3     return x + y;
4 }
```

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:

```
1 int addInteger(int x, int y)
2 {
3     return x + y;
4 }
5
6 double addDouble(double x, double y)
7 {
8     return x + y;
9 }
```

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:

```
1 double add(double x, double y)
2 {
3     return x + y;
4 }
```

We now have two version of `add()`:

```
1 int add(int x, int y); // integer version
2 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:

```
1 int add(int x, int y, int z)
2 {
3     return x + y + z;
4 }
```

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:

```
1 int getRandomValue();
2 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:

```
1 int getRandomInt();
2 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).

```
1 void getRandomValue(int &out);
2 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:

```
1 // method 1: getRandomInt() returns an int
2 printValue(getRandomInt()); // easy
3
4 // method 2: getRandomValue() has an int out parameter
5 int temp; // now we need a temp variable
6 getRandomValue(temp); // to call getRandomValue(int) here
7 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:

```
1 typedef char *string;
2 void print(string value);
3 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:

```
1 void print(char *value);
2 void print(int value);
3
4 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:

```
1 void print(char *value);
2 void print(int value);
3
4 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:

```
1 struct Employee; // defined somewhere else
2 void print(float value);
3 void print(Employee value);
4
5 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.

```
1 class X; // with user-defined conversion to int
2
3 void print(float value);
```

```

4 void print(int value);
5
6 X value; // declare a variable named value of type class X
7 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:

```

1 void print(unsigned int value);
2 void print(float value);
3
4 print('a');
5 print(0);
6 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:

```

1 int x = 0;
2 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:

```

1 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 [4.12 -- 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:

```
1  #include <iostream>
2
3  void fcn(char c, int x)
4  {
5      std::cout << 'a';
6  }
7
8  void fcn(char c, double x)
9  {
10     std::cout << 'b';
11 }
12
13 void fcn(char c, float x)
14 {
15     std::cout << 'c';
16 }
17
18 int main()
19 {
20     fcn('x', 4);
21 }
```

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 arguments



Index



7.5 -- Inline functions

 C++ TUTORIAL |  PRINT THIS POST

106 comments to 7.6 — Function overloading

[« Older Comments](#) [1](#) [2](#)



Napalm

August 31, 2019 at 4:34 am · Reply

1. I wanted to check that you still need separate function declarations when the parameters are different but return type is the same.

2. For small functions, is this the right way to organise files? With separate .h and .cpp? Or would you put the definitions in the .h file?

Functions.h

```
1  #ifndef FUNCTIONS_H
2  #define FUNCTIONS_H
3
4
5  #include <Windows.h>
6  #include <iostream>
7  #include <string>
8
9
10 // Prints string output slowly, where time is the time
11 // between characters in milliseconds
12 void printStr(std::string str, int time);
13 void printStr(std::string str);
14
15
16 #endif
```

Functions.cpp

```
1  #include "Functions.h"
2
3  void printStr(std::string str, int time)
4  {
5      for (char c : str)
6      {
```

```
7         std::cout << c;  
8         Sleep(time);  
9     }  
10 }  
11  
12 void printStr(std::string str)  
13 {  
14     std::cout << str;  
15 }
```

**nascardriver**August 31, 2019 at 11:27 pm · Reply.

- Pass/return non-fundamental types by reference.
- Don't `#include <Windows.h>`. Your code won't work on other OSs.
- Include as few headers as possible in headers. `Functions.h` doesn't need `<iostream>`.

Yes that's how you'd separate it. Don't define functions in headers (There are exceptions about which you'll learn later).

**Napalm**September 1, 2019 at 2:14 am · Reply.

Do I `#include <iostream>` in `Functions.cpp` then? For some reason I thought that was wrong. Maybe I'm mixing that up with including .cpp files themselves, which I'm pretty sure is wrong.

**nascardriver**September 1, 2019 at 2:16 am · Reply.

> Do I `#include <iostream>` in `Functions.cpp` then?
Yes. "`Functions.cpp`" is the file using functionality from `<iostream>`, so it should be including it.

**Napalm**September 1, 2019 at 3:38 am · Reply.

So, just to be clear, if a .h file requires `std::<string>`, you still `#include <string>` in the .h file, but not in the corresponding .cpp, due to contents being copied over?

**nascardriver**September 1, 2019 at 3:45 am · Reply.

Correct. In some situations I'll still include `<string>` in the .cpp file, even if I already have access through the header. Do what works best for you.

**Napalm**September 1, 2019 at 3:51 am · Reply.

Gotcha, thanks very much!



Nirbhay

[August 1, 2019 at 4:06 am · Reply](#)

Hi!

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);
```

I do not understand the use of '*' here. Please explain.

Thanks :)



nascar driver

[August 1, 2019 at 4:12 am · Reply](#)

It's a pointer/[char array]
Lesson 6.6 to 6.8a



Nirbhay

[August 1, 2019 at 4:18 am · Reply](#)

typedef char *string is same as using *string = char.

so can we write

void print(*string value) = void print(char value) ?

I do not understand how the parameters 'string value' and 'char *value' are equal.



nascar driver

[August 1, 2019 at 4:20 am · Reply](#)

```
1 | using string = char*;
```



Nirbhay

[August 1, 2019 at 4:29 am · Reply](#)

Alright.

Is it a valid assignment, if a string is passed as an argument to the function: void print(char *value)?



nascar driver

[August 1, 2019 at 4:30 am · Reply](#)

Yes, `string` is the same as `char*`.

**Nirbhay**[August 1, 2019 at 4:33 am · Reply](#)

Thank you so much. I am sure, to the beginners char* is frightening.

**potterman28wxcv**[August 17, 2019 at 8:54 am · Reply](#)

If they read the previous lessons it shouldn't be frightening since the author explains in the linked lessons what pointers are, and why C strings are stored as (char *).

**hassan magaji**[February 5, 2019 at 9:46 am · Reply](#)

thanks alot nas...

**hassan magaji**[February 3, 2019 at 1:34 am · Reply](#)

hi Alex,

i've seen some using function-like macros and deprecating function overloading

like this:

```
1 | #define add(x,y) ((x)+(y)) // being much more GENERIC
```

is this advisable anyway?

...ready to stick to your rules/recommendations always...

**nascardriver**[February 3, 2019 at 5:55 am · Reply](#)

Quoting lesson 2.10:

"Function-like macros act like functions, and serve a similar purpose. We will not discuss them here, because their use is generally considered dangerous, and almost anything they can do can be done by a normal function."

**DecSco**[July 26, 2019 at 8:24 am · Reply](#)

And if you need that functionality, you should look into templates.

[« Older Comments](#)[1](#) [2](#)