11.2 — Function overload differentiation

ALEX[1] DECEMBER 11, 2024

In the prior lesson (11.1 -- Introduction to function overloading (https://www.learncpp.com/cpp-tutorial/introduction-to-function-overloading/)[2]), we introduced the concept of function overloading, which allows us to create multiple functions with the same name, so long as each identically named function has different parameter types (or the functions can be otherwise differentiated).

In this lesson, we'll take a closer look at how overloaded functions are differentiated. Overloaded functions that are not properly differentiated will cause the compiler to issue a compile error.

How overloaded functions are differentiated

Function property	Used for differentiation	Notes
Number of parameters	Yes	
Type of parameters	Yes	Excludes typedefs, type aliases, and const qualifier on value parameters. Includes ellipses.
Return type	No	

Note that a function's return type is not used to differentiate overloaded functions. We'll discuss this more in a bit.

For advanced readers

For member functions, additional function-level qualifiers are also considered:

Function-level qualifier	Used for overloading
const or volatile	Yes
Ref-qualifiers	Yes

As an example, a const member function can be differentiated from an otherwise identical non-const member function (even if they share the same set of parameters).

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/)[3].

Overloading based on number of parameters

An overloaded function is differentiated so long as each overloaded function has a different number of parameters. For example:

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

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

The compiler can easily tell that a function call with two integer parameters should go to `add(int, int)` and a function call with three integer parameters should go to `add(int, int, int)`.

Overloading based on type of parameters

A function can also be differentiated so long as each overloaded function's list of parameter types is distinct. For example, all of the following overloads are differentiated:

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

Because type aliases (or typedefs) are not distinct types, overloaded functions using type aliases are not distinct from overloads using the aliased type. For example, all of the following overloads are not differentiated (and will result in a compile error):

```cpp
typedef int Height; // typedef
using Age = int; // type alias

void print(int value);
void print(Age value); // not differentiated from print(int)
void print(Height value); // not differentiated from print(int)
```

For parameters passed by value, the const qualifier is also not considered. Therefore, the following functions are not considered to be differentiated:

```cpp
void print(int);
void print(const int); // not differentiated from print(int)
```

For advanced readers

We haven't covered ellipsis yet, but ellipsis parameters are considered to be a unique type of parameter:

```cpp
void foo(int x, int y);
void foo(int x, ...); // differentiated from foo(int, int)
```

Thus a call to `foo(4, 5)` will match to `foo(int, int)`, not `foo(int, ...)`.

The return type of a function is not considered for differentiation

A function's return type is not considered when differentiating overloaded functions.

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:

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

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

```
error C2556: 'double getRandomValue(void)': overloaded function differs only by re
```

This makes sense. If you were the compiler, and you saw this statement:

```cpp
getRandomValue();
```

Which of the two overloaded functions would you call? It's not clear.

As an aside…

This was an intentional choice, as it ensures the behavior of a function call 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 in the function call. If return values were used for differentiation, then we wouldn't have an easy syntactic way to tell which overload 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.

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

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

Type signature

A function's **type signature** (generally called a **signature**) is defined as the parts of the function header that are used for differentiation of the function. In C++, this includes the function name, number of parameters, parameter type, and function-level qualifiers. It notably does *not* include the return type.

Name mangling

As an aside…

When the compiler compiles a function, it performs **name mangling**, which means the compiled name of the function is altered ("mangled") based on various criteria, such as the number and type of parameters, so that the linker has unique names to work with.

For example, a function with prototype `int fcn()` might compile to mangled name `__fcn_v`, whereas `int fcn(int)` might compile to mangled name `__fcn_i`. So while in the source code, the two overloaded functions share the name `fcn()`, in compiled code, the mangled names are unique (`__fcn_v` vs `__fcn_i`).

There is no standardization on how names should be mangled, so different compilers will produce different mangled names.

Next lesson
11.3 Function overload resolution and ambiguous matches

4

Back to table of contents

5

Previous lesson
11.1 Introduction to function overloading

2

6

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[8] are connected to your provided email address.

56 COMMENTS

Newest

junky
May 14, 2025 1:41 am PDT

// Online C++ compiler to run C++ program online
#include <iostream>

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

int add(int x, int y, int z)
{
std::cout << "second add" << '\n';
return x + y + z;
}

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

int main()
{
std::cout << add(1, 2);
std::cout << '\n';
std::cout << add(1.2, 3.4);
std::cout << '\n';
std::cout << add(1.2, 5.4, 9.9);

return 0;
}

I'm still confused as to how this is possible, even though I had double arguments here add(1.2, 5.4, 9.9) since there was no double with 3 params then it defaulted to use the int function

0 Reply

Eric
Reply to junky 9 May 28, 2025 10:23 am PDT

The way I understand it, you have a function add that takes 3 integer parameters:

```cpp
int add(int x, int y, int z)
{
std::cout << "second add" << '\n';
return x + y + z;
}
```

When you called add() with 3 double parameters, the compiler searches for a function named "add" and finds the 3 you've defined earlier. Then it compares number of parameters and finds one that matches. Your function definition having parameters of type int just means the parameters of type double you passed to it in main() are converted to type int, discarding the fractional parts. So running your program will output 15 for the last call to add.

1 Reply

Dinny
February 26, 2025 3:10 am PST

I'm looking forward to reading about how overloads work with optional parameters in one of the upcoming lessons :)

0 Reply

yzdpw
February 8, 2025 11:25 pm PST

Complier: "I am confused!"

3 Reply

sa00
January 16, 2025 7:55 am PST

Wow I'm learning a lot of stuff. I come from a background in Python, but C++ really teaches you more about computers. Thanks Alex for the amazing tutorials!

4 Reply

rkh
December 9, 2024 9:08 am PST

Hi
`So while in the source code, two overloaded functions share a name, in compiled code, the names are actually unique.`

What do the names mean here, exact or compiled names? The compiled names of two overloaded functions must be differentiated to prevent compile errors. am I right?

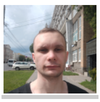
0 Reply

Alex Author
Reply to rkh 10 December 11, 2024 10:05 pm PST

Rewrote the sentence as "So while in the source code, the two overloaded functions share the name `fcn()`, in compiled code, the mangled names are unique (`__fcn_v` vs `__fcn_i`)."

The mangled names in the compiled code must be unique to prevent linker errors (otherwise the linker will think we've defined the same function more than once, which violates the ODR).

0 Reply



Vitalik
July 26, 2024 10:39 am PDT

Interesting question:
void print(int);
void print(const int); // not differentiated from print(int)

void is a method.
And functions are called functions that can return something as a result, aren't they?

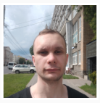
I've noticed that everyone writes them differently. Alex, I would love to hear your opinion on this matter.

1 Reply

Alex Author
Reply to Vitalik 11 July 26, 2024 4:37 pm PDT

`void` is a type. In C++, we don't use the term "method" -- we use "function". Functions can be value returning or not.

1 Reply



Vitalik
Reply to Alex 12 July 26, 2024 5:05 pm PDT

Thank you!

1 Reply

wizwiz
February 7, 2024 5:09 pm PST

Some helpful resources about the name mangling:

1. You can compile files with `gcc -c filename.cpp -o filename.o`
2. afterwards, one could look into the .o file with `objdump -D filename.o > instructions.txt`
3. To demangle names that appear in instructions.txt file one could use c++filt command as such: `c++filt __Z3addid`

this would produce the following output: add(int, double)

13 Reply

zls
November 6, 2023 7:39 am PST

By function level qualifiers do you mean const, constexpr?

0 Reply

Alex Author

Reply to zls [13] November 6, 2023 4:30 pm PST

A const member function has a function level const qualifier. e.g. `int getValue() const`

Functions can't be differentiated on constexpr-ness.

1 Reply

James

August 23, 2023 5:20 pm PDT

```
type aliases (or typedefs) are not distinct types

typedef int Height; // typedef
using Age = int; // type alias

void print(int value);
void print(Age value); // not differentiated from print(int)
void print(Height value); // not differentiated from print(int)
```

would it be okay if I used aliases/typedefs but it's in different types? Like:

```
typedef double DoubleNumber;
using LongNumb = long;

void print(int value);
void print(DoubleNumber value);
void print(LongNumb value);
```

1 Reply

Alex Author

Reply to James [14] August 28, 2023 12:43 pm PDT

Sure, but you may run into problems later. For example, if you later add:

```
using BigCount = long;
void print(BigCount value);
```

This will cause a compiler error because `void print(LongNumb value);` and `void print(BigCount value);` will both resolve to `void print(long value);` and the compiler will treat this as a duplicate definition.

2 Reply

Ali
August 16, 2023 4:17 am PDT

can you make an example of `type signature`?
What kind of signature ?

0 Reply

Teretana
Reply to Ali [15] December 31, 2024 4:50 pm PST

someFcn(int,double)
someFcn()
someFcn(double, double, double)
const somefcn (int, double)
const someFcn ()
volatile someFcn ()

Should all be different signatures for someFcn.
Would love to be corrected if i m wrong.

0 Reply

Alex Author
Reply to Ali [15] August 17, 2023 9:16 pm PDT

From the lesson: "A function's type signature is defined as the parts of the function header that are used for differentiation of the function. In C++, this includes the function name, number of parameter, parameter type, and function-level qualifiers. It notably does not include the return type."

What part of this isn't clear?

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/introduction-to-function-overloading/
3. https://www.learncpp.com/cpp-tutorial/ellipsis-and-why-to-avoid-them/
4. https://www.learncpp.com/cpp-tutorial/function-overload-resolution-and-ambiguous-matches/
5. https://www.learncpp.com/
6. https://www.learncpp.com/function-overload-differentiation/
7. https://www.learncpp.com/cpp-tutorial/type-deduction-for-functions/
8. https://gravatar.com/
9. https://www.learncpp.com/cpp-tutorial/function-overload-differentiation/#comment-610033

10. https://www.learncpp.com/cpp-tutorial/function-overload-differentiation/#comment-605044
11. https://www.learncpp.com/cpp-tutorial/function-overload-differentiation/#comment-600103
12. https://www.learncpp.com/cpp-tutorial/function-overload-differentiation/#comment-600147
13. https://www.learncpp.com/cpp-tutorial/function-overload-differentiation/#comment-589484
14. https://www.learncpp.com/cpp-tutorial/function-overload-differentiation/#comment-586000
15. https://www.learncpp.com/cpp-tutorial/function-overload-differentiation/#comment-585636
16. https://g.ezoic.net/privacy/learncpp.com