8.10 — Function overload differentiation

👤 ALEX   🕐 OCTOBER 4, 2021

In the prior lesson (8.9 -- Introduction to function overloading), 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 parameters (or the functions can be otherwise differentiated).

In this lesson, we'll talk about 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

The easiest way to differentiate an overload function is to ensure each overloaded function has a different set (number and/or type) of parameters.

Function property	Used for overloading	Notes
Number of parameters	Yes	
Type of parameters	Yes	Excludes typedefs, type aliases, and const qualifier on value paramet
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).

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 set of parameter types are 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_t; // typedef
using age_t = int; // type alias

void print(int value);
void print(age_t value); // not differentiated from print(int)
void print(height_t 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)
```

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:

```
1  int getRandomValue();
   double
2  getRandomValue();
```

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

```
error C2556: 'double getRandomValue(void)': overloaded function differs only by return type from 'int getRan
domValue(void)'
```

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

```
1  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:

```
1  int getRandomInt();
   double
2  getRandomDouble();
```
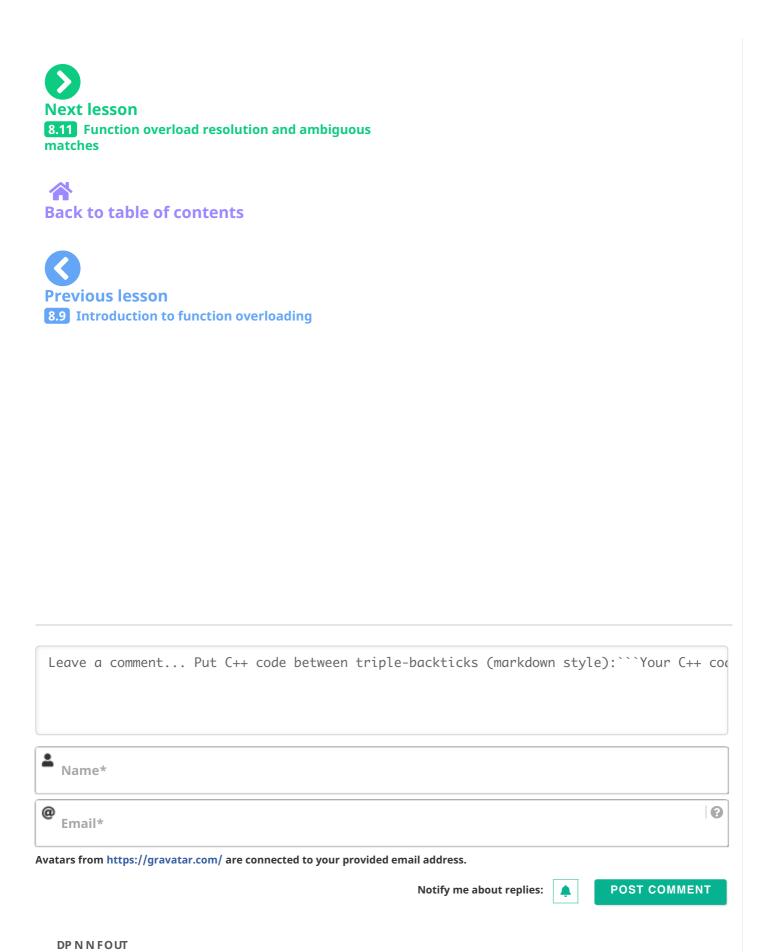
Name mangling

> **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, some function with prototype `int fcn()` might compile to name `__fcn_v`, whereas `int fcn(int)` might compile to name `__fcn_i`. So while in the source code, two overloaded functions share a name, in compiled code, the names are actually unique.
>
> There is no standardization on how names should be mangled, so different compilers will produce different mangled names.

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ co

Name*

Email*

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

Notify me about replies:

POST COMMENT

DP N N F OUT

Newest