8.9 — Introduction to function overloading

ALEX JUNE 18, 2021

Consider the following function:

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

This trivial function adds two integers and returns an integer result. However, what if we also want a function that can add two floating point numbers? This `add()` function is not 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:

```cpp
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 function naming standard for similar functions that have parameters of different types, remember the names of these functions, and actually call the correct one.

And then what happens when we want to have a similar function that adds 3 integers instead of 2? Managing unique names for each function quickly becomes burdensome.

Introduction to function overloading

Fortunately, C++ has an elegant solution to handle such cases. Function overloading 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). Each function sharing a name (in the same scope) is called an overloaded function (sometimes called anoverload for short).

To overload our `add()` function, we can simply declare another `add()` function that takes double parameters:

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

We now have two versions of `add()` in the same scope:

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

double add(double x, double y) // floating point
version
{
    return x + y;
}

int main()
{
    return 0;
}
```

The above program will compile. Although you might expect these functions to result in a naming conflict, that is not the case here. Because the parameter types of these functions differ, the compiler is able to differentiate these functions, and will treat them as separate functions that just happen to share a name.

Key insight

Functions can be overloaded so long as each overloaded function can be differentiated by the compiler. If an overloaded function can not be differentiated, a compile error will result.

Related content

Because operators in C++ are just functions, operators can also be overloaded. We'll discuss this in 13.1 -- Introduction to operator overloading.

Introduction to overload resolution

Additionally, when a function call is made to function that has been overloaded, the compiler will try to match the function call to the appropriate overload based on the arguments used in the function call. This is called overload resolution.

Here's a simple example demonstrating this:

```
#include <iostream>

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

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

int main()
{
    std::cout << add(1, 2); // calls add(int, int)
    std::cout << '\n';
    std::cout << add(1.2, 3.4); // calls add(double, double)

    return 0;
}
```

The above program compiles and produces the result:

```
3
4.6
```

When we provide integer arguments in the call to `add(1, 2)`, the compiler will determine that we're trying to call `add(int, int)`. And when we provide floating point arguments in the call to `add(1.2, 3.4)`, the compiler will determine that we're trying to call `add(double, double)`.

Making it compile

In order for program using overloaded functions to compile, two things have to be true:

1. Each overloaded function has to be differentiated from the others. We discuss how functions can be differentiated in lesson 8.10 -- Function overload differentiation.
2. Each call to an overloaded function has to resolve to an overloaded function. We discuss how the compiler matches function calls to overloaded functions in lesson 8.11 -- Function overload resolution and ambiguous matches.

If an overloaded function is not differentiated, or if a function call to an overloaded function can not be resolved to an overloaded function, then a compile error will result.

In the next lesson, we'll explore how overloaded functions can be differentiated from each other. Then, in the following lesson, we'll explore how the compiler resolves function calls to overloaded functions.

Conclusion

Function overloading provides a great way to reduce the complexity of your program by reducing the number of function names you need to remember. It can and should be used liberally.

Best practice

Use function overloading to make your program simpler.

Next lesson
8.10 Function overload differentiation

Back to table of contents

Previous lesson
8.8 Type deduction for functions

X

X