11.4 — Deleting functions

learncpp.com/cpp-tutorial/deleting-functions/

In some cases, it is possible to write functions that don't behave as desired when called with values of certain types.

Consider the following example:

```
#include <iostream>

void printInt(int x)
{
    std::cout << x << '\n';
}

int main()
{
    printInt(5);    // okay: prints 5
    printInt('a');  // prints 97 -- does this make sense?
    printInt(true); // print 1 -- does this make sense?

    return 0;
}
```

This example prints:

```
5
97
1
```

While `printInt(5)` is clearly okay, the other two calls to `printInt()` are more questionable. With `printInt('a')`, the compiler will determine that it can promote `'a'` to int value `97` in order to match the function call with the function definition. And it will promote `true` to int value `1`. And it will do so without complaint.

Let's assume we don't think it makes sense to call `printInt()` with a value of type `char` or `bool`. What can we do?

Deleting a function using the `= delete` specifier

In cases where we have a function that we explicitly do not want to be callable, we can define that function as deleted by using the **= delete** specifier. If the compiler matches a function call to a deleted function, compilation will be halted with a compile error.

Here's an updated version of the above making use of this syntax:

```
#include <iostream>

void printInt(int x)
{
    std::cout << x << '\n';
}

void printInt(char) = delete; // calls to this function will halt compilation
void printInt(bool) = delete; // calls to this function will halt compilation

int main()
{
    printInt(97);   // okay

    printInt('a');  // compile error: function deleted
    printInt(true); // compile error: function deleted

    printInt(5.0);  // compile error: ambiguous match

    return 0;
}
```

Let’s take a quick look at some of these. First, `printInt('a')` is a direct match for `printInt(char)`, which is deleted. The compiler thus produces a compilation error. `printInt(true)` is a direct match for `printInt(bool)`, which is deleted, and thus also produces a compilation error.

`printInt(5.0)` is an interesting case, with perhaps unexpected results. First, the compiler checks to see if exact match `printInt(double)` exists. It does not. Next, the compiler tries to find a best match. Although `printInt(int)` is the only non-deleted function, the deleted functions are still considered as candidates in function overload resolution. Because none of these functions are unambiguously the best match, the compiler will issue an ambiguous match compilation error.

Key insight

`= delete` means “I forbid this”, not “this doesn’t exist”.

Deleted function participate in all stages of function overload resolution (not just in the exact match stage). If a deleted function is selected, then a compilation error results.

For advanced readers

Other types of functions can be similarly deleted.

We discuss deleting member functions in lesson 14.14 -- Introduction to the copy constructor, and deleting function template specializations in lesson 11.7 -- Function template instantiation.

Deleting all non-matching overloads Advanced

Deleting a bunch of individual function overloads works fine, but can be verbose. There may be times when we want a certain function to be called only with arguments whose types exactly match the function parameters. We can do this by using a function template (introduced in upcoming lesson 11.6 -- Function templates) as follows:

```cpp
#include <iostream>

// This function will take precedence for arguments of type int
void printInt(int x)
{
    std::cout << x << '\n';
}

// This function template will take precedence for arguments of other types
// Since this function template is deleted, calls to it will halt compilation
template <typename T>
void printInt(T x) = delete;

int main()
{
    printInt(97);   // okay
    printInt('a');  // compile error
    printInt(true); // compile error

    return 0;
}
```