11.4 — Deleting functions

👤 **ALEX**[1]   🕐 **DECEMBER 28, 2023**

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:

```cpp
#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:

```
 1   #include <iostream>
 2
 3   void printInt(int x)
 4   {
 5       std::cout << x << '\n';
 6   }
 7
 8   void printInt(char) = delete; // calls to this function will halt compilation
 9   void printInt(bool) = delete; // calls to this function will halt compilation
10
11   int main()
12   {
13       printInt(97);    // okay
14
15       printInt('a');  // compile error: function deleted
16       printInt(true); // compile error: function deleted
17
18       printInt(5.0);  // compile error: ambiguous match
19
20       return 0;
21   }
```

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 (https://www.learncpp.com/cpp-tutorial/introduction-to-the-copy-constructor/)[2], and deleting function template specializations in lesson 11.7 -- Function template instantiation (https://www.learncpp.com/cpp-tutorial/function-template-instantiation/)[3].

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 (https://www.learncpp.com/cpp-tutorial/function-templates/)[4]) 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;
}
```

5

Back to table of contents

6

7

8

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

Leave a comment...

Name*

@ Email*   t

Find a mistake? Leave a comment above!

Notify me about replies:

POST COMMENT

29 COMMENTS 👥⚙

Newest ▾

XenophonSoulis
🕓 January 22, 2025 9:25 am PST

This is a bit nitpicky, but isn't 1 the desired result when you std::cout a bool? So in the line

```
1 | printInt(true); // print 1 -- does this make sense?
```

the printed result is the one we wanted anyway? I think a std::boolalpha would make the example easier to understand.

👍 0   ➤ Reply

Manas Ghosh
🕓 November 12, 2024 10:58 am PST

in the 2nd example, there's only one printInt function. then how does overload resolution come into picture?

👍 0   ➤ Reply

> **Alex** [Author]
> 💬 Reply to Manas Ghosh [12] 🕓 November 14, 2024 1:43 pm PST
>
> No, there are three. Two of them are deleted. The deleted ones still count.
>
> 👍 0   ➤ Reply

KLAP
🕓 October 18, 2024 2:05 pm PDT

with multiple parameters, it seems like it only works if the order parameter's type is exactly match which is convenient to use.

```
1    #include <iostream>
2
3    void printInt(int x, char y, double z)
4    {
5        std::cout << x << static_cast<char>(y) << z << '\n';
6    }
7
8    void printInt(int, double, char) = delete;
9    void printInt(char, int, double) = delete;
10   void printInt(double, char, int) = delete;
11   void printInt(char, double, int) = delete;
12   int main()
13   {
14       printInt(97, '9', 9.4); //This still works
15       return 0;
16   }
```

Last edited 8 months ago by KLAP

👍 0   ➦ Reply

Not quite. The best matching function is determined as per usual. If the resulting best match function is deleted, then you get a compilation error.

The best matching function doesn't have to be an exact match -- it just has to be unambiguously the best match.

👍 2   ➦ Reply

User0
🕐 October 7, 2024 5:06 am PDT

```
1    #include <iostream>
2    void f(int x) {std::cout << x;}
3    void f(char) = delete;
4
5    int main() {
6      f(1.1);
7    }
```

This code throws an ambiguous error, as you said, but I'm still not sure if I understand the reasoning of this.

1. I call an `f(double)`.
2. The compiler tries to look after `f(double)`, it doesn't see it.
3. The compiler tries to find the best match, there's `f(int)`, while there's also a deleted function `f(char)`.
4. Now it throws an ambiguous error, since the compiler still considers both `f(int)` and `f(char)` to match against `f(double)`.

If I call a `f(bool)`, this error doesn't occur, but if I call a `f(double)` it does.

If I call `f(bool)`, there would be: `f(bool)` -> `f(int)` or `f(char)`; there wouldn't be an exact match, then it would try numeric promotion, which would match `f(int)` and unambiguously succeed to match it and return me no error (as it did successfully)

If I call `f(double)`, there would be: `f(bool)` -> `f(int)` or `f(char)`; there wouldn't be an exact match, then it would try numeric promotion, which would happen, then there would be numeric conversion, I would have both `f(int)` and `f(char)` to match, which would be an ambiguous match, giving me the error.

Is it the right explanation? The one in this lesson let me wondering what the ambiguity would really mean.

✎ *Last edited 8 months ago by User0*

👍 0   ↪ Reply
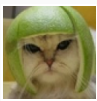
Alex `Author`
💬 Reply to User0 [14]   🕐 October 9, 2024 10:00 am PDT

Yes, you basically have it right. f(double) matches both f(int) and f(char) via conversion. This is an ambiguous match since the compiler can't tell which one we prefer.

👍 1   ↪ Reply

Saanidhya
🕐 June 7, 2024 10:43 am PDT

What if the arguments to the function are not known till runtime, such as when inputs and outputs are involved?

👍 0   ↪ Reply

Alex `Author`
💬 Reply to Saanidhya [15]   🕐 June 9, 2024 6:45 pm PDT

The types of the arguments to the function must be known at compile time. The values don't matter in this context.

👍 3   ↪ Reply

MOEGMA25
💬 Reply to Saanidhya [15]   🕐 June 9, 2024 9:44 am PDT

I tried first using:

```cpp
#include <iostream>

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

int add(int, char) = delete;

int main()
{
    int a{};
    std::cin >> a;
    std::cout << add(4, a);

    return 0;
}
```

input: g

output: 4

But a char (as well as text) will just result to the value 0 (explained in a prior lesson). Let's try something else:

```cpp
#include <iostream>

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

int add(int, double) = delete;

int main()
{
    int a{};
    std::cin >> a;
    std::cout << add(4, a);

    return 0;
}
```

input: 2.5

output: 6

Now we have frictions dropped.

Maybe this?

```cpp
1    #include <iostream>
2
3    int add(int x, int y)
4    {
5        return x + y;
6    }
7
8    int add(int, double) = delete;
9
10   int main()
11   {
12       int a{};
13       std::cin >> a;
14
15       std::cout << add(4, static_cast<double>(a));
16
17       return 0;
18   }
```

Compile error.

Guess arithmetic conversions just take over to prevent a wrong argument from being passed on ¯_(ツ)_/¯

(Alex please clearify if wrong)

👍 0   ➤ Reply

Strain

💬 Reply to MOEGMA25 [16]   🕐 June 10, 2024 7:06 am PDT

`std::cin` will ignore any semantically nonsensical input according to the type.

In the first example, "g" doesn't mean anything for `int`, so it gets ignored (ergo, `a` continues as `0` from the `{}`). In the second one, `std::cin` grabs just the "2", as the "." is nonsensical for `int`, and the ".5" is just left in `std::cin`.

There is a reason for why it's called a *compiler error*.

👍 1   ➤ Reply
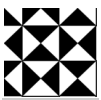
Andy B

🕐 May 29, 2024 10:14 am PDT

Does it matter what the type of the deleted function is?

Best practice to match the type? Best to use void?

```cpp
1    int addOne(int a){return a + 1;}
2
3    void addOne(char a) = delete; //void? int? just pick one and don't bother thinking
     about it?
```

✏️ *Last edited 1 year ago by Andy B*

👍 0   ➤ Reply

Alex Author

⟲ Reply to Andy B [17] ⏱ May 29, 2024 4:41 pm PDT

From the compiler's perspective, the specific return type provided doesn't matter since it's not used in overload resolution.

Best practice is to provide whatever return type you normally would. Otherwise, if you did something like this: `std::cout << addOne('c')` your compiler would probably generate extra errors if you had a `void` return type.

👍 1 ➤ Reply

Jestin

⏱ March 25, 2024 6:22 am PDT

why this code not working ?
when I disable
`void printFloat(char, char) = delete; // calls to this function will halt compilation`
it works.

```cpp
#include <iostream>

void printFloat(float x, ...)
{
    std::cout << x << '\n';
}

void printFloat(char, char) = delete; // calls to this function will halt compilation

int main()
{
    printFloat(9.7f, 7.8f); // prints 9.7

    // printFloat('a','b');  // compile error: function deleted

    return 0;
}
```

👍 0 ➤ Reply

Alex Author

⟲ Reply to Jestin [18] ⏱ March 25, 2024 12:16 pm PDT

Because your overloaded functions are ambiguous. When resolving overloads, a standard conversion is considered a better match than a match to ellipses. So the top function has an exact match and ellipses match. The bottom function has two standard conversion matches. Since the first argument is a better match to the top function and the second argument is a better match to the bottom function, the compiler can't tell which should take precedence.

👍 2 ➤ Reply

Jestin

yes, this works as expected.

```cpp
#include <iostream>

void printFloat(float x, float y, ...)
{
    std::cout << x << '\n';
    std::cout << y << '\n';
}

void printFloat(char, char, ...) = delete;

int main()
{
    printFloat(9.7f, 7.8f, 8.9f);

    return 0;
}
```

./Output

9.7

7.8

✎ Last edited 1 year ago by Jestin

👍 0   → Reply

Ali
March 6, 2024 5:08 am PST

Is this kind of halt an abnormal termination? Does doing this result in skipping garbage collection?

👍 0   → Reply

Alex Author

In this lesson, we're talking about halting compilation, not a runtime halt.

👍 1   → Reply

Ali
March 6, 2024 5:05 am PST

In case the best match is deleted, a compilation error will raise:

```cpp
void f(double) = delete;
//...
f(0.0f); // err
```

However this one will be ambiguous:

```
1  void f(float) = delete;
2  ///...
3  f(0.0); // ambiguous
```

👍 1   ➦ Reply

zls
🕐 December 9, 2023 11:58 pm PST

```
1   void printInt(int x) {
2       std::cout << x << '\n';
3   }
4
5   void printInt(char) = delete;
6   void printInt(bool) = delete;
7
8   int main() {
9       printInt(5);
10      printInt(5.0f);
11  }
```

Why calling printInt(float) results in an ambiguous match, there is only one function call it can match to. The other overloads are deleted, why does it count them also in overload resolution.

👍 2   ➦ Reply

Alex Author
💬 Reply to zls [21]   🕐 December 11, 2023 11:36 am PST

Deleted functions participate in all stages of function overload resolution. Presumably this is to help keep things consistent, as it would be inconsistent for deleted functions to only be considered for some types of matches (e.g. exact matches and promotion matches) and not others (e.g. matches requiring conversions)

👍 4   ➦ Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/introduction-to-the-copy-constructor/
3. https://www.learncpp.com/cpp-tutorial/function-template-instantiation/
4. https://www.learncpp.com/cpp-tutorial/function-templates/
5. https://www.learncpp.com/cpp-tutorial/default-arguments/
6. https://www.learncpp.com/

7. https://www.learncpp.com/cpp-tutorial/function-overload-resolution-and-ambiguous-matches/
8. https://www.learncpp.com/deleting-functions/
9. https://www.learncpp.com/cpp-tutorial/ref-qualifiers/
10. https://www.learncpp.com/cpp-tutorial/the-conditional-operator/
11. https://gravatar.com/
12. https://www.learncpp.com/cpp-tutorial/deleting-functions/#comment-604104
13. https://www.learncpp.com/cpp-tutorial/deleting-functions/#comment-603320
14. https://www.learncpp.com/cpp-tutorial/deleting-functions/#comment-602803
15. https://www.learncpp.com/cpp-tutorial/deleting-functions/#comment-598104
16. https://www.learncpp.com/cpp-tutorial/deleting-functions/#comment-598175
17. https://www.learncpp.com/cpp-tutorial/deleting-functions/#comment-597698
18. https://www.learncpp.com/cpp-tutorial/deleting-functions/#comment-595085
19. https://www.learncpp.com/cpp-tutorial/deleting-functions/#comment-595099
20. https://www.learncpp.com/cpp-tutorial/deleting-functions/#comment-594355
21. https://www.learncpp.com/cpp-tutorial/deleting-functions/#comment-590722
22. https://g.ezoic.net/privacy/learncpp.com