

## 12.10 — The hidden “this” pointer

ALEX SEPTEMBER 2, 2021

One of the questions about classes that new object-oriented programmers often ask is, “When a member function is called, how does C++ keep track of which object it was called on?”. The answer is that C++ utilizes a hidden pointer named “this”! Let’s take a look at “this” in more detail.

The following is a simple class that holds an integer and provides a constructor and access functions. Note that no destructor is needed because C++ can clean up integer member variables for us.

```
1 class Simple
2 {
3     private:
4         int m_id;
5     public:
6         Simple(int id)
7             : m_id{ id }
8         {
9         }
10        void setID(int id) { m_id =
11        id; }
12        int getID() { return m_id; }
13    };
```

Here’s a sample program that uses this class:

```
1 int main()
2 {
3     Simple simple{1};
4     simple.setID(2);
5     std::cout << simple.getID() <<
6     '\n';
7     return 0;
8 }
```

As you would expect, this program produces the result:

```
2
```

Somehow, when we call `simple.setID(2);`, C++ knows that function `setID()` should operate on object `simple`, and that `m_id` actually refers to `simple.m_id`. Let’s examine the mechanics behind how this works.

---

## The hidden \*this pointer

Take a look at the following line of code from the example above:

```
1 | simple.setID(2);
```

Although the call to function `setID()` looks like it only has one argument, it actually has two! When compiled, the compiler converts `simple.setID(2);` into the following:

```
1 | setID(&simple, 2); // note that simple has been changed from an object prefix to a function argument!
```

Note that this is now just a standard function call, and the object `simple` (which was formerly an object prefix) is now passed by address as an argument to the function.

But that's only half of the answer. Since the function call now has an added argument, the member function definition needs to be modified to accept (and use) this argument as a parameter. Consequently, the following member function:

```
1 | void setID(int id) { m_id =  
  | id; }
```

is converted by the compiler into:

```
1 | void setID(Simple* const this, int id) { this->m_id =  
  | id; }
```

When the compiler compiles a normal member function, it implicitly adds a new parameter to the function named "this". The `this` pointer is a hidden `const` pointer that holds the address of the object the member function was called on.

There's just one more detail to take care of. Inside the member function, any class members (functions and variables) also need to be updated so they refer to the object the member function was called on. This is easily done by adding a "this->" prefix to each of them. Thus, in the body of function `setID()`, `m_id` (which is a class member variable) has been converted to `this->m_id`. Thus, when "this" points to the address of `simple`, `this->m_id` will resolve to `simple.m_id`.

Putting it all together:

1. When we call `simple.setID(2)`, the compiler actually calls `setID(&simple, 2)`.
2. Inside `setID()`, the "this" pointer holds the address of object `simple`.
3. Any member variables inside `setID()` are prefixed with "this->". So when we say `m_id = id`, the compiler is actually executing `this->m_id = id`, which in this case updates `simple.m_id` to `id`.

The good news is that all of this happens automatically, and it doesn't really matter whether you remember how it works or not. All you need to remember is that all normal member functions have a "this" pointer that refers to the object the function was called on.

---

## "this" always points to the object being operated on

New programmers are sometimes confused about how many "this" pointers exist. Each member function has a "this" pointer parameter that is set to the address of the object being operated on. Consider:

```

1  int main()
2  {
3      Simple A{1}; // this = &A inside the Simple
        constructor
        Simple B{2}; // this = &B inside the Simple
        constructor
        A.setID(3); // this = &A inside member function
        setID
4      B.setID(4); // this = &B inside member function
        setID

        return 0;
    }

```

Note that the “this” pointer alternately holds the address of object A or B depending on whether we’ve called a member function on object A or B.

Because “this” is just a function parameter, it doesn’t add any memory usage to your class (just to the member function call, since that parameter needs to be passed to the function and stored in memory).

## Explicitly referencing “this”

Most of the time, you never need to explicitly reference the “this” pointer. However, there are a few occasions where doing so can be useful:

First, if you have a constructor (or member function) that has a parameter with the same name as a member variable, you can disambiguate them by using “this”:

```

1  class Something
2  {
3  private:
4      int data;
5  public:
6      Something(int data)
7      {
8          this->data = data; // this->data is the member, data is the local
            parameter
9      }
10 };

```

Note that our constructor is taking a parameter of the same name as a member variable. In this case, “data” refers to the parameter, and “this->data” refers to the member variable. Although this is acceptable coding practice, we find using the “m\_” prefix on all member variable names provides a better solution by preventing duplicate names altogether!

Some developers prefer to explicitly add this-> to all class members. We recommend that you avoid doing so, as it tends to make your code less readable for little benefit. Using the m\_ prefix is a more readable way to differentiate member variables from non-member (local) variables.

## Chaining member functions

Second, it can sometimes be useful to have a class member function return the object it was working with as a return value. The primary reason to do this is to allow a series of member functions to be “chained” together, so several member functions can be called on the same object! You’ve actually been doing this for a long time. Consider this common example where you’re outputting more than one bit of text using std::cout:

```
1 | std::cout << "Hello, " <<
   | userName;
```

In this case, `std::cout` is an object, and `operator<<` is a member function that operates on that object. The compiler evaluates the above snippet like this:

```
1 | (std::cout << "Hello, ") <<
   | userName;
```

First, `operator<<` uses `std::cout` and the string literal `"Hello, "` to print `"Hello, "` to the console. However, since this is part of an expression, `operator<<` also needs to return a value (or `void`). If `operator<<` returned `void`, you'd end up with this:

```
1 | (void) <<
   | userName;
```

which clearly doesn't make any sense (and the compiler would throw an error). Instead, `operator<<` returns `*this`, which in this context is the `std::cout` object. That way, after the first `operator<<` has been evaluated, we get:

```
1 | (std::cout) <<
   | userName;
```

which then prints the user's name.

In this way, we only need to specify the object (in this case, `std::cout`) once, and each function call passes it on to the next function to work with, allowing us to chain multiple commands together.

We can implement this kind of behavior ourselves. Consider the following class:

```
1 | class Calc
2 | {
3 | private:
4 |     int m_value{0};
5 | public:
6 |
7 |     void add(int value) { m_value += value;
8 | }
9 |     void sub(int value) { m_value -= value;
   | }
   |     void mult(int value) { m_value *=
   | value; }
   |
   |     int getValue() { return m_value; }
   | };
```

If you wanted to add 5, subtract 3, and multiply by 4, you'd have to do this:

```

1 #include <iostream>
2 int main()
3 {
4     Calc calc{};
5     calc.add(5); // returns void
6     calc.sub(3); // returns void
7     calc.mult(4); // returns void
8
9     std::cout << calc.getValue() <<
10    '\n';
11    return 0;
12 }

```

However, if we make each function return `*this`, we can chain the calls together. Here is the new version of `Calc` with “chainable” functions:

```

1 class Calc
2 {
3 private:
4     int m_value{};
5
6 public:
7     Calc& add(int value) { m_value += value; return *this; }
8     Calc& sub(int value) { m_value -= value; return *this; }
9     Calc& mult(int value) { m_value *= value; return
10    *this; }
11
12     int getValue() { return m_value; }
13 };

```

Note that `add()`, `sub()` and `mult()` are now returning `*this`. Consequently, this allows us to do the following:

```

1 #include <iostream>
2 int main()
3 {
4     Calc calc{};
5     calc.add(5).sub(3).mult(4);
6
7     std::cout << calc.getValue() <<
8     '\n';
9     return 0;
10 }

```

We have effectively condensed three lines into one expression! Let’s take a closer look at how this works.

First, `calc.add(5)` is called, which adds 5 to our `m_value`. `add()` then returns `*this`, which is just a reference to `calc`, so `calc` will be the object used in the subsequent evaluation. Next `calc.sub(3)` evaluates, which subtracts 3 from `m_value` and again returns `calc`. Finally, `calc.mult(4)` multiplies `m_value` by 4 and returns `calc`, which isn’t used further, and is thus ignored.

Since each function modified `calc` as it was executed, `calc`’s `m_value` now contains the value  $((0 + 5) - 3) * 4$ , which is 8.

## Summary

The “`this`” pointer is a hidden parameter implicitly added to any non-static member function. Most of the time, you will not need to

access it directly, but you can if needed. It's worth noting that "this" is a const pointer -- you can change the value of the underlying object it points to, but you can not make it point to something else!

By having functions that would otherwise return void return \*this instead, you can make those functions chainable. This is most often used when overloading operators for your classes (something we'll talk about more in [chapter 13](#)).



## Next lesson

**12.11** Class code and header files



[Back to table of contents](#)



## Previous lesson

**12.9** Destructors

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



Name\*



Email\*



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

Notify me about replies:



POST COMMENT

DP N N FOUT

Newest ▼

