# 8.8 — The hidden “this” pointer

BY ALEX ON SEPTEMBER 6TH, 2007 | LAST MODIFIED BY ALEX ON JUNE 29TH, 2018

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.

```
class Simple
{
private:
    int m_id;

public:
    Simple(int id)
    {
        setID(id);
    }

    void setID(int id) { m_id = id; }
    int getID() { return m_id; }
};
```

Here’s a sample program that uses this class:

```
int main()
{
    Simple simple(1);
    simple.setID(2);
    std::cout << simple.getID() << '\n';

    return 0;
}
```

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:

```
    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:

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

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:

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

is converted by the compiler into:

```
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 transparently to you as a programmer, 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:

```
int main()
{
    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
    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 goes on the stack while the function is executing).

**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”:

```
class Something
{
private:
    int data;

public:
    Something(int data)
    {
        this->data = data;
    }
};
```

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.

*Recommendation: Do not add this-> to all uses of your class members. Only do so when you have a specific reason to.*

**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:

```
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:

```
(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 need to return a value (or void). If operator<< returned void, you'd end up with this:

```
(void) << userName;
```

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

```
(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:

```
class Calc
{
private:
    int m_value;

public:
    Calc() { m_value = 0; }

    void add(int value) { m_value += value; }
    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:

```
#include <iostream>
int main()
{
    Calc calc;
    calc.add(5); // returns void
    calc.sub(3); // returns void
    calc.mult(4); // returns void

    std::cout << calc.getValue() << '\n';
    return 0;
}
```

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

```
class Calc
{
private:
    int m_value;

public:
    Calc() { m_value = 0; }

    Calc& add(int value) { m_value += value; return *this; }
    Calc& sub(int value) { m_value -= value; return *this; }
    Calc& mult(int value) { m_value *= value; return *this; }

    int getValue() { return m_value; }
};
```

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

```
#include <iostream>
int main()
{
    Calc calc;
    calc.add(5).sub(3).mult(4);

    std::cout << calc.getValue() << '\n';
    return 0;
}
```

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 a reference to calc. Our expression is now calc.sub(3).mult(4). calc.sub(3) subtracts 3 from m_value and returns calc. Our expression is now calc.mult(4). calc.mult(4) multiplies m_value by 4 and returns calc, which is then ignored. However, 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 9).



**8.9 -- Class code and header files**


**Index**


**8.7 -- Destructors**