15.1 — The hidden “this” pointer and member function chaining

One of the questions about classes that new programmers often ask is, “When a member function is called, how does C++ keep track of which object it was called on?”.

First, let’s define a simple class to work with. This class encapsulates an integer value, and provides some access functions to get and set that value:

```
#include <iostream>

class Simple
{
private:
    int m_id{};

public:
    Simple(int id)
        : m_id{ id }
    {
    }

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

    void print() const { std::cout << m_id; }
};

int main()
{
    Simple simple{1};
    simple.setID(2);

    simple.print();

    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`.

The answer is that C++ utilizes a hidden pointer named `this`! In this lesson, we'll take a look at `this` in more detail.

The hidden `this` pointer

Inside every member function, the keyword **this** is a const pointer that holds the address of the current implicit object.

Most of the time, we don't mention `this` explicitly, but just to prove we can:

```
#include <iostream>

class Simple
{
private:
    int m_id{};

public:
    Simple(int id)
        : m_id{ id }
    {
    }

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

    void print() const { std::cout << this->m_id; } // use `this` pointer to access
the implicit object and operator-> to select member m_id
};

int main()
{
    Simple simple{ 1 };
    simple.setID(2);

    simple.print();

    return 0;
}
```

This works identically to prior example, and prints:

```
2
```

Note that the `print()` member functions from the prior two examples do exactly the same thing:

```
void print() const { std::cout << m_id; }       // implicit use of this
void print() const { std::cout << this->m_id; } // explicit use of this
```

It turns out that the former is shorthand for the latter. When we compile our programs, the compiler will implicitly prefix any member referencing the implicit object with `this->`. This helps keep our code more concise and prevents the redundancy from having to explicitly write `this->` over and over.

A reminder

We use `->` to select a member from a pointer to an object. `this->m_id` is the equivalent of `(*this).m_id`.

We cover `operator->` in lesson 13.12 -- Member selection with pointers and references.

How is `this` set?

Let's take a closer look at this function call:

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

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

```
Simple::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 also needs to be modified to accept (and use) this argument as a parameter. Here's our original member function definition for `setID()`:

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

How the compiler rewrites functions is an implementation-specific detail, but the end-result is something like this:

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

Note that our `setId` function has a new leftmost parameter named `this`, which is a const pointer (meaning it cannot be re-pointed, but the contents of the pointer can be modified). The `m_id` member has also been rewritten as `this->m_id`, utilizing the `this` pointer.

For advanced readers

In this context, the `static` keyword means the function is not associated with objects of the class, but instead is treated as if it were a normal function inside the scope region of the class. We cover static member functions in lesson 15.7 -- Static member functions.

Putting it all together:

1. When we call `simple.setID(2)`, the compiler actually calls `Simple::setID(&simple, 2)`, and `simple` is passed by address to the function.
2. The function has a hidden parameter named `this` which receives the address of `simple`.
3. Member variables inside setID() are prefixed with `this->`, which points to `simple`. So when the compiler evaluates `this->m_id`, it's actually resolving to `simple.m_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 non-static member functions have a `this` pointer that refers to the object the function was called on.

Key insight

All non-static member functions have a `this` const pointer that holds the address of the implicit object.

`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 single `this` pointer parameter that points to the implicit object. 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 (and not a member), it does not make instances of your class larger memory-wise.

Explicitly referencing `this`

Most of the time, you won't need to explicitly reference the `this` pointer. However, there are a few occasions where doing so can be useful:

First, if you have a member function that has a parameter with the same name as a data member, you can disambiguate them by using `this`:

```
struct Something
{
    int data{}; // not using m_ prefix because this is a struct

    void setData(int data)
    {
        this->data = data; // this->data is the member, data is the local parameter
    }
};
```

This `Something` class has a member named `data`. The function parameter of `setData()` is also named `data`. Within the `setData()` function, `data` refers to the function parameter (because the function parameter shadows the data member), so if we want to reference the `data` member, we use `this->data`.

Some developers prefer to explicitly add `this->` to all class members to make it clear that they are referencing a member. 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 concise way to differentiate private member variables from non-member (local) variables.

Returning `*this`

Second, it can sometimes be useful to have a member function return the implicit object as a return value. The primary reason to do this is to allow member functions to be "chained" together, so several member functions can be called on the same object in a single expression! This is called **function chaining** (or **method chaining**).

Consider this common example where you're outputting several bits of text using `std::cout`:

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

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 needs to return a value (or `void`). If `operator<<` returned `void`, you'd end up with this as the partially evaluated expression:

```
void{} << userName;
```

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

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

which then prints the user's name.

This way, we only need to specify `std::cout` once, and then we can chain as many pieces of text together using `operator<<` as we want. Each call to `operator<<` returns `std::cout` so the next call to `operator<<` uses `std::cout` as the left operand.

We can implement this kind of behavior in our member functions too. Consider the following class:

```
class Calc
{
private:
    int m_value{};

public:

    void add(int value) { m_value += value; }
    void sub(int value) { m_value -= value; }
    void mult(int value) { m_value *= value; }

    int getValue() const { 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` by reference, we can chain the calls together. Here is the new version of `Calc` with "chainable" functions:

```
class Calc
{
private:
    int m_value{};

public:
    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() const { return m_value; }
};
```

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

```
#include <iostream>

int main()
{
    Calc calc{};
    calc.add(5).sub(3).mult(4); // method chaining

    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 `m_value`. `add()` then returns `*this`, which is just a reference to implicit object `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, the `m_value` of `calc` now contains the value (((0 + 5) - 3) * 4), which is `8`.

This is probably the most common explicit use of `this`, and is one you should consider whenever it makes sense to have chainable member functions.

Because `this` always points to the implicit object, we don't need to check whether it is a null pointer before dereferencing it.

Resetting a class back to default state

If your class has a default constructor, you may be interested in providing a way to return an existing object back to its default state.

As noted in previous lessons (14.12 -- Delegating constructors), constructors are only for initialization of new objects, and should not be called directly. Doing so will result in unexpected behavior.

The best way to reset a class back to a default state is to create a `reset()` member function, have that function create a new object (using the default constructor), and then assign that new object to the current implicit object, like this:

```
void reset()
{
    *this = {}; // value initialize a new object and overwrite the implicit object
}
```

Here's a full program demonstrating this `reset()` function in action:

```
#include <iostream>

class Calc
{
private:
    int m_value{};

public:
    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() const { return m_value; }

    void reset() { *this = {}; }
};


int main()
{
    Calc calc{};
    calc.add(5).sub(3).mult(4);

    std::cout << calc.getValue() << '\n'; // prints 8

    calc.reset();

    std::cout << calc.getValue() << '\n'; // prints 0

    return 0;
}
```

`this` and const objects

For non-const member functions, `this` is a const pointer to a non-const value (meaning `this` cannot be pointed at something else, but the object pointing to may be modified). With const member functions, `this` is a const pointer to a const value (meaning the pointer cannot be pointed at something else, nor may the object being pointed to be modified).

The errors generated from attempting to call a non-const member on a const object can be a little cryptic:

```
error C2662: 'int Something::getValue(void)': cannot convert 'this' pointer from
'const Something' to 'Something &'
error: passing 'const Something' as 'this' argument discards qualifiers [-
fpermissive]
```

When we call a non-const member function on a const object, the implicit `this` function parameter is a const pointer to a *non-const* object. But the argument has type const pointer to a *const* object. Converting a pointer to a const object into a pointer to a non-const object requires discarding the const qualifier, which cannot be done implicitly. The compiler error generated by some compilers reflects the compiler complaining about being asked to perform such a conversion.

Why `this` a pointer and not a reference

Since the `this` pointer always points to the implicit object (and can never be a null pointer unless we've done something to cause undefined behavior), so you may be wondering why `this` is a pointer instead of a reference. The answer is simple: when `this` was added to C++, references didn't exist yet.

If `this` were added to the C++ language today, it would undoubtedly be a reference instead of a pointer. In other more modern C++-like languages, such as Java and C#, `this` is implemented as a reference.