

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

👤 ALEX¹ ⌚ DECEMBER 29, 2024

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:

```
1  #include <iostream>
2
3  class Simple
4  {
5  private:
6      int m_id{};
7
8  public:
9      Simple(int id)
10         : m_id{ id }
11     {
12     }
13
14     int getID() const { return m_id; }
15     void setID(int id) { m_id = id; }
16
17     void print() const { std::cout << m_id; }
18 };
19
20 int main()
21 {
22     Simple simple{1};
23     simple.setID(2);
24
25     simple.print();
26
27     return 0;
28 }
```

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:

```
1  #include <iostream>
2
3  class Simple
4  {
5  private:
6      int m_id{};
7
8  public:
9      Simple(int id)
10         : m_id{ id }
11     {
12     }
13
14     int getID() const { return m_id; }
15     void setID(int id) { m_id = id; }
16
17     void print() const { std::cout << this->m_id; } // use `this` pointer to access
the implicit object and operator-> to select member m_id
18 };
19
20 int main()
21 {
22     Simple simple{ 1 };
23     simple.setID(2);
24
25     simple.print();
26
27     return 0;
28 }
```

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:

```
1  void print() const { std::cout << m_id; } // implicit use of this
2  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](https://www.learncpp.com/cpp-tutorial/member-selection-with-pointers-and-references/) (<https://www.learncpp.com/cpp-tutorial/member-selection-with-pointers-and-references/>)².

How is `this` set?

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

```
1 | 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:

```
1 | 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()`:

```
1 | 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:

```
1 | 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](https://www.learncpp.com/cpp-tutorial/static-member-functions/) (<https://www.learncpp.com/cpp-tutorial/static-member-functions/>)³.

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:

```
1 int main()
2 {
3     Simple a{1}; // this = &a inside the Simple constructor
4     Simple b{2}; // this = &b inside the Simple constructor
5     a.setID(3); // this = &a inside member function setID()
6     b.setID(4); // this = &b inside member function setID()
7
8     return 0;
9 }
```

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`:

```
1 struct Something
2 {
3     int data{}; // not using m_ prefix because this is a struct
4
5     void setData(int data)
6     {
7         this->data = data; // this->data is the member, data is the local parameter
8     }
9 };
```

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`:

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

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 as the partially evaluated expression:

```
1 | 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:

```
1 | (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:

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

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

```

1  #include <iostream>
2
3  int main()
4  {
5      Calc calc{};
6      calc.add(5); // returns void
7      calc.sub(3); // returns void
8      calc.mult(4); // returns void
9
10     std::cout << calc.getValue() << '\n';
11
12     return 0;
13 }

```

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:

```

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 *this; }
10
11     int getValue() const { return m_value; }
12 };

```

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

```

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

```

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 a reference to `*this`, which is 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](https://www.learncpp.com/cpp-tutorial/delegating-constructors/) (<https://www.learncpp.com/cpp-tutorial/delegating-constructors/>)⁴), 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:

```
1 void reset()
2 {
3     *this = {}; // value initialize a new object and overwrite the implicit object
4 }
```

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

```
1 #include <iostream>
2
3 class Calc
4 {
5 private:
6     int m_value{};
7
8 public:
9     Calc& add(int value) { m_value += value; return *this; }
10    Calc& sub(int value) { m_value -= value; return *this; }
11    Calc& mult(int value) { m_value *= value; return *this; }
12
13    int getValue() const { return m_value; }
14
15    void reset() { *this = {}; }
16 };
17
18
19 int main()
20 {
21     Calc calc{};
22     calc.add(5).sub(3).mult(4);
23
24     std::cout << calc.getValue() << '\n'; // prints 8
25
26     calc.reset();
27
28     std::cout << calc.getValue() << '\n'; // prints 0
29
30     return 0;
31 }
```

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 function on a const object can be a little cryptic:

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

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), 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.



[Next lesson](#)

15.2 [Classes and header files](#)

5



[Back to table of contents](#)

6



[Previous lesson](#)

14.x [Chapter 14 summary and quiz](#)

7

8



B

U

URL

INLINE CODE

C++ CODE BLOCK

HELP!

Leave a comment...

 Name*

@ Email*




Notify me about replies:



POST COMMENT

 Find a mistake? Leave a comment above!?

 Avatars from <https://gravatar.com/>¹⁰ are connected to your provided email address.

351 COMMENTS

Newest ▼



Call Me John

🕒 June 8, 2025 1:28 am PDT

Hi Alex, in your program example at the very top, what is the function of getID() if it's not used?

```
#include <iostream>
```

```
class Simple
```

```
{
```

```
private:
```

```
int m_id{};
```

```
public:
```

```
Simple(int id)
```

```
: m_id{ id }
```

```
{
```

```
}
```

```
int getID() const { return m_id; } // what the function of this line?
```

```
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;
```

```
}
```

👍 0 ➡ Reply

 **Mr. F**
🕒 May 25, 2025 3:40 pm PDT

its all good and dandy until he starts talking about const... "this" is terrible :p

👍 0 ➡ Reply

 **剑指漠北**
🕒 May 22, 2025 3:59 am PDT

There is another way to not use the this pointer

```
1  #include <iostream>
2
3  struct Something
4  {
5      int data{};
6
7      void setData(int data)
8      {
9          //this->data = data;
10         Something::data = data;
11     }
12 };
13
14 int main()
15 {
16
17     Something something;
18     something.data = 10;
19     std::cout << something.data << "\n";
20     something.setData(20);
21     std::cout << something.data << "\n";
22
23     return 0;
24 }
```

output:

10

20

But when I misspelled it, the compiler didn't complain.

I am very confused, please help me

```

1  #include <iostream>
2  #include <optional>
3  #include <vector>
4
5  struct Something
6  {
7      int data{};
8
9      void setData(int data)
10     {
11         //this->data = data;
12         Something::data = data; // here, : (not ::)
13     }
14 };
15
16 int main()
17 {
18
19     Something something;
20     something.data = 10;
21     std::cout << something.data << "\n";
22     something.setData(20);
23     std::cout << something.data << "\n";
24
25     return 0;
26 }

```

output:

10

10

another

This pointer is also used when a non-static member function passes a class pointer or class reference to an external function or variable.

```

1  #include <iostream>
2
3  struct Something;
4  void fun(Something* something);
5  void fun(const Something& something);
6
7  Something* gSomethingPtr = nullptr;
8
9  struct Something
10 {
11     int data{};
12
13     void setData(int data)
14     {
15         this->data = data;
16     }
17     int getData() const {
18         return data;
19     }
20
21     void doSomething()
22     {
23         gSomethingPtr = this;
24         fun(this);
25         fun(*this);
26     }
27 };
28
29 void fun(Something* something) {
30     std::cout << "fun(Something* something)\n";
31     something->setData(200);
32 }
33
34 void fun(const Something& something) {
35     std::cout << "fun(const Something& something)\n";
36     std::cout << "data:" << something.getData() << "\n";
37 }
38
39 int main()
40 {
41     Something something;
42     something.data = 10;
43     std::cout << something.data << "\n";
44     something.setData(20);
45     std::cout << something.data << "\n";
46     something.doSomething();
47
48     return 0;
49 }

```

output:

10

20

fun(Something* something)

fun(const Something& something)

fun:200

Using *this in constructor or destructor may cause semi-construction or semi-destruction problems. Passing this to the outside of the class in a constructor or destructor may also cause semi-construction or semi-destruction problems.

👍 2 ➡ Reply



vitrums

🕒 April 23, 2025 12:48 am PDT

```
1 | *this = {}; // value initialize a new object and overwrite the implicit object
```

There's so much involved in this line: move semantics, copying operator, destructor call. And yet the wording in the comment provided above implies something silly as `this = &(Calc{})`. Obviously neither address-of operator won't work on a temp, nor is `this` mutable. But I still find it funny that this seemingly harmless paragraph isn't marked as "For advanced readers".

👍 0 ➡ Reply



剑指漠北

🗨 Reply to [vitrums](#)¹¹ 🕒 May 26, 2025 7:13 pm PDT

How about this code:

```
1 | void reset()
2 | {
3 |     // *this = {};
4 |     this->~Calc(); // explicat destructor call
5 |     new(this)Calc(); // placement new
6 | }
```

The above code only has destructor and constructor calls

📝 Last edited 1 month ago by 剑指漠北

👍 0 ➡ Reply



Leni

🕒 March 9, 2025 9:31 pm PDT

The concept of the hidden `this` pointer in C++ simplifies object-oriented programming by implicitly keeping track of which object a member function is acting upon, allowing for more concise code. How does returning `*this` by reference enable method chaining, and in what scenarios would it be most beneficial in real-world applications?

👍 0 ➡ Reply



NordicCat

🕒 January 1, 2025 9:50 pm PST

Do you agree on this??

<https://javascript.info/ninja-code>

Actually, I am coming from js. Should we write ninja code??

1 Reply



Alex Author

Reply to NordicCat¹² January 6, 2025 3:41 pm PST

I agree that it's an attempt at humor.

4 Reply



vivien

October 2, 2024 12:29 pm PDT

Why can't I use this as a default argument in a member function? Like this

```
void add(int value, Calc* c=this)
```

Where Calc is the class

4 Reply



Alex Author

Reply to vivien¹³ October 3, 2024 10:26 am PDT

Per https://en.cppreference.com/w/cpp/language/default_arguments: "The this pointer is not allowed in default arguments".

3 Reply



Jay

September 17, 2024 3:07 am PDT

"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." When is the high level const applied to the argument? If the argument is generated via `obj.foo(* &obj *)` wouldn't that nameless pointer argument be a non-const pointer to a const/non-const object?

0 Reply



Alex Author

Reply to Jay¹⁴ September 20, 2024 1:40 pm PDT

You can't call a non-const member function on a const object.

`this` for a non-const member function is a "const pointer to non-const".

`this` for a const member function is a "const pointer to const".

`&obj` for a non-const object is a "non-const pointer to non-const".

`&obj` for a const object is a "non-const pointer to const".

The type of an argument must match the type of the parameter. If it does not, the compiler will try to implicitly convert it to match. This conversion can add a top-level and/or low-level const (but not remove it).

Therefore:

`&obj` for a non-const object ("non-const pointer to non-const") can convert to `this` for a non-const member function ("const pointer to non-const").

`&obj` for a non-const object ("non-const pointer to non-const") can convert to `this` for a const member function ("const pointer to const").

`&obj` for a const object ("non-const pointer to const") can convert to `this` for a const member function ("const pointer to const").

`&obj` for a const object ("non-const pointer to const") can NOT convert to `this` for a non-const member function ("const pointer to non-const"), as this requires discarding a const.

👍 0

↩ Reply



Joyboy

🕒 September 16, 2024 2:37 am PDT

lets a class foo exists and we create a class object, `foo x{...}`; so the "this" pointer is exactly the same as `foo* const ptr{&x}`; am i correct in saying this?

👍 0

↩ Reply



Alex

Author

↩ Reply to [Joyboy](#)¹⁵ 🕒 September 20, 2024 10:44 am PDT

Within a member function invoked on `x`, yes.

👍 0

↩ Reply



Arifin Depay Jr

🕒 August 18, 2024 9:44 pm PDT

```
class Point
{
private:
int m_x{1};
int m_y{1};
public:
Point() = default;
Point(Point* const A , int x , int y)
: m_x{x} /* cant use copy initialization */
, m_y{y}
{
// or A->m_x = x
// or A->m_y = y
}
void print(Point* const A)
{
std::cout << A->m_x << "\t" << A->m_y << "\n";
}
```

```
Point& return_(Point* const A)
```

```
{  
    return *A;  
}
```

```
Point& return_reset(Point* const A)
```

```
{  
    return A = {}; //set it to default //which is m_x{1}, m_y{1} */  
}  
};
```

Since I cant use "this" as a parameter(Compiler implicit this for us)
so I use "A" to represent "this"

 0  Reply

Links

1. <https://www.learncpp.com/author/Alex/>
2. <https://www.learncpp.com/cpp-tutorial/member-selection-with-pointers-and-references/>
3. <https://www.learncpp.com/cpp-tutorial/static-member-functions/>
4. <https://www.learncpp.com/cpp-tutorial/delegating-constructors/>
5. <https://www.learncpp.com/cpp-tutorial/classes-and-header-files/>
6. <https://www.learncpp.com/>
7. <https://www.learncpp.com/cpp-tutorial/chapter-14-summary-and-quiz/>
8. <https://www.learncpp.com/the-hidden-this-pointer-and-member-function-chaining/>
9. <https://www.learncpp.com/cpp-tutorial/destructors/>
10. <https://gravatar.com/>
11. <https://www.learncpp.com/cpp-tutorial/the-hidden-this-pointer-and-member-function-chaining/#comment-609488>
12. <https://www.learncpp.com/cpp-tutorial/the-hidden-this-pointer-and-member-function-chaining/#comment-606080>
13. <https://www.learncpp.com/cpp-tutorial/the-hidden-this-pointer-and-member-function-chaining/#comment-602638>
14. <https://www.learncpp.com/cpp-tutorial/the-hidden-this-pointer-and-member-function-chaining/#comment-602056>
15. <https://www.learncpp.com/cpp-tutorial/the-hidden-this-pointer-and-member-function-chaining/#comment-602020>
16. <https://g.ezoic.net/privacy/learncpp.com>

