12.13 — In and out parameters

ALEX[1] DECEMBER 9, 2024

A function and its caller communicate with each other via two mechanisms: parameters and return values. When a function is called, the caller provides arguments, which the function receives via its parameters. These arguments can be passed by value, reference, or address.

Typically, we'll pass arguments by value or by const reference. But there are times when we may need to do otherwise.

In parameters

In most cases, a function parameter is used only to receive an input from the caller. Parameters that are used only for receiving input from the caller are sometimes called **in parameters**.

```
#include <iostream>

void print(int x) // x is an in parameter
{
    std::cout << x << '\n';
}

void print(const std::string& s) // s is an in parameter
{
    std::cout << s << '\n';
}

int main()
{
    print(5);
    std::string s { "Hello, world!" };
    print(s);

    return 0;
}
```

In-parameters are typically passed by value or by const reference.

Out parameters

A function argument passed by non-const reference (or by pointer-to-non-const) allows the function to modify the value of an object passed as an argument. This provides a way for a function to return data back to the caller in cases where using a return value is not sufficient for some reason.

A function parameter that is used only for the purpose of returning information back to the caller is called an **out parameter**.

For example:

```cpp
#include <cmath>    // for std::sin() and std::cos()
#include <iostream>

// sinOut and cosOut are out parameters
void getSinCos(double degrees, double& sinOut, double& cosOut)
{
    // sin() and cos() take radians, not degrees, so we need to convert
    constexpr double pi { 3.14159265358979323846 }; // the value of pi
    double radians = degrees * pi / 180.0;
    sinOut = std::sin(radians);
    cosOut = std::cos(radians);
}

int main()
{
    double sin { 0.0 };
    double cos { 0.0 };

    double degrees{};
    std::cout << "Enter the number of degrees: ";
    std::cin >> degrees;

    // getSinCos will return the sin and cos in variables sin and cos
    getSinCos(degrees, sin, cos);

    std::cout << "The sin is " << sin << '\n';
    std::cout << "The cos is " << cos << '\n';

    return 0;
}
```

This function has one parameter `degrees` (whose argument is passed by value) as input, and "returns" two parameters (by reference) as output.

We've named these out parameters with the suffix "out" to denote that they're out parameters. This helps remind the caller that the initial value passed to these parameters doesn't matter, and that we should expect them to be overwritten. By convention, output parameters are typically the rightmost parameters.

Let's explore how this works in more detail. First, the main function creates local variables `sin` and `cos`. Those are passed into function `getSinCos()` by reference (rather than by value). This means function `getSinCos()` has access to the actual `sin` and `cos` variables in `main()`, not just copies. `getSinCos()` accordingly assigns new values to `sin` and `cos` (through references `sinOut` and `cosOut` respectively), which overwrites the old values in `sin` and `cos`. Function `main()` then prints these updated values.

If `sin` and `cos` had been passed by value instead of reference, `getSinCos()` would have changed copies of `sin` and `cos`, leading to any changes being discarded at the end of the function. But because `sin` and `cos` were passed by reference, any changes made to `sin` or `cos` (through the references) are persisted beyond the function. We can therefore use this mechanism to return values back to the caller.

As an aside…

This answer on StackOverflow (https://stackoverflow.com/a/9779765)[2] is an interesting read that explains why non-const lvalue references are not allowed to bind to rvalues/temporary objects (due to implicit type conversion producing unexpected behavior when combined with out-parameters).

Out parameters have an unnatural usage syntax

Out-parameters, while functional, have a few downsides.

First, the caller must instantiate (and initialize) objects and pass them as arguments, even if it doesn't intend to use them. These objects must be able to be assigned to, which means they can't be made const.

Second, because the caller must pass in objects, these values can't be used as temporaries, or easily used in a single expression.

The following example shows both of these downsides:

```cpp
#include <iostream>

int getByValue()
{
    return 5;
}

void getByReference(int& x)
{
    x = 5;
}

int main()
{
    // return by value
    [[maybe_unused]] int x{ getByValue() }; // can use to initialize object
    std::cout << getByValue() << '\n';      // can use temporary return value in
expression

    // return by out parameter
    int y{};                // must first allocate an assignable object
    getByReference(y);      // then pass to function to assign the desired value
    std::cout << y << '\n'; // and only then can we use that value

    return 0;
}
```

As you can see, the syntax for using out-parameters is a bit unnatural.

Out-parameters by reference don't make it obvious the arguments will be modified

When we assign a function's return value to an object, it is clear that the value of the object is being modified:

```cpp
x = getByValue(); // obvious that x is being modified
```

This is good, as it makes it clear that we should expect the value of `x` to change.

However, let's take a look at the function call to `getSinCos()` in the example above again:

```cpp
getSinCos(degrees, sin, cos);
```

It is not clear from this function call that `degrees` is an in parameter, but `sin` and `cos` are out-parameters. If the caller does not realize that `sin` and `cos` will be modified, a semantic error will likely result.

Using pass by address instead of pass by reference can in some case help make out-parameters more obvious by requiring the caller to pass in the address of objects as arguments.

Consider the following example:

```
void foo1(int x);  // pass by value
void foo2(int& x); // pass by reference
void foo3(int* x); // pass by address

int main()
{
    int i{};

    foo1(i);  // can't modify i
    foo2(i);  // can modify i (not obvious)
    foo3(&i); // can modify i

    int *ptr { &i };
    foo3(ptr); // can modify i (not obvious)

    return 0;
}
```

Notice that in the call to `foo3(&i)`, we have to pass in `&i` rather than `i`, which helps make it clearer that we should expect `i` to be modified.

However, this is not fool-proof, as `foo3(ptr)` allows `foo3()` to modify `i` and does not require the caller to take the address-of `ptr`.

The caller may also think they can pass in `nullptr` or a null pointer as a valid argument when this is disallowed. And the function is now required to do null pointer checking and handling, which adds more complexity. This need for added null pointer handling often causes more issues than just sticking with pass by reference.

For all of these reasons, out-parameters should be avoided unless no other good options exist.

Best practice

Avoid out-parameters (except in the rare case where no better options exist).

Prefer pass by reference for non-optional out-parameters.

In/out parameters

In rare cases, a function will actually use the value of an out-parameter before overwriting its value. Such a parameter is called an **in-out parameter**. In-out-parameters work identically to out-parameters and have all the same challenges.

When to pass by non-const reference

If you're going to pass by reference in order to avoid making a copy of the argument, you should almost always pass by const reference.

Author's note

In the following examples, we will use `Foo` to represent some type that we care about. For now, you can imagine `Foo` as a type alias for a type of your choice (e.g. `std::string`).

However, there are two primary cases where pass by non-const reference may be the better choice.

First, use pass by non-const reference when a parameter is an in-out-parameter. Since we're already passing in the object we need back out, it's often more straightforward and performant to just modify that object.

```cpp
void someFcn(Foo& inout)
{
    // modify inout
}

int main()
{
    Foo foo{};
    someFcn(foo); // foo modified after this call, may not be obvious

    return 0;
}
```

Giving the function a good name can help:

```cpp
void modifyFoo(Foo& inout)
{
    // modify inout
}

int main()
{
    Foo foo{};
    modifyFoo(foo); // foo modified after this call, slightly more obvious

    return 0;
}
```

The alternative is to pass the object by value or const reference instead (as per usual) and return a new object by value, which the caller can then assign back to the original object:

```cpp
Foo someFcn(const Foo& in)
{
    Foo foo { in }; // copy here
    // modify foo
    return foo;
}

int main()
{
    Foo foo{};
    foo = someFcn(foo); // makes it obvious foo is modified, but another copy made
here

    return 0;
}
```

This has the benefit of using a more conventional return syntax, but requires making 2 extra copies (sometimes the compiler can optimize one of these copies away).

Second, use pass by non-const reference when a function would otherwise return an object by value to the caller, but making a copy of that object is *extremely* expensive. Especially if the function is called many times in a performance-critical section of code.

```cpp
void generateExpensiveFoo(Foo& out)
{
    // modify out
}

int main()
{
    Foo foo{};
    generateExpensiveFoo(foo); // foo modified after this call

    return 0;
}
```

For advanced readers

The most common example of the above is when a function needs to fill a large C-style array or `std::array` with data, and the array has an expensive-to-copy element type. We discuss arrays in a future chapter.

That said, objects are rarely so expensive to copy that resorting to non-conventional methods of returning those objects is worthwhile.

Next lesson
12.14 Type deduction with pointers, references, and const

3

Back to table of contents

4

Previous lesson
12.12 Return by reference and return by address

5

6

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/[9] are connected to your provided email address.

82 COMMENTS

Newest

Jeyser
March 31, 2025 9:55 am PDT

As a learning project, I made a space invaders clone game where I passed 20x10 matrix "map" (array of arrays) as a reference to different functions, which then updated the matrix (player and enemy position etc.). Was this a justifiable use of out parameters or is there a better way to do this? Thank you for the amazing tutorials, I have learned a lot in just few weeks!

Last edited 3 months ago by Jeyser

0 Reply

SinixND
January 24, 2025 1:53 am PST

I'd like to mention thee points:

(Repetition of my Reply to `MOEGMA25` January 24, 2025)

1. What is your opinion on the following? (Assuming `Foo` being a big datatype)

The intention is to avoid potentially expensive copies of big datatypes `Foo`, but also ensure (optionally: enforce via `[[nodiscard]]`) clear communication of the modification:

```
[[nodiscard]] // Optional, enforces clear communication
Foo const& costlyModification(Foo& bigParameterReference)
{
    // modify big parameter here
    return bigParameterReference;
}

int main()
{
    Foo bigData{};
    bigData = costlyModification(bigData); // Communicates modification clearly
}
```

2. I tried to find a commonly used convention for ordering parameters in regards to in/out/io-usage.

One point that influenced my decision to order `Output` before `InOut` before `Input` is that one would have to make an exception for parameters with default values, which need to be the rightmost/last ones.

`Output` and `InOut` parameters can not reasonably have default values, but sole input parameters definitely can.

3. Personally I fully agree with your example on pointer usage. I don't like it, because you would have to convert a pointer to a reference if you wanted to keep the obvious usage for tramp/pass-through data:

```
// Obviously, this function is not for practical use :D
void func(Foo* ptr) // Function takes pointer parameter to communicate that it will be modified
{
    func(ptr); // Pointer pass-through/forwarding, non obvious modification; You mentioned this.

    Foo& ref(&ptr); // Possible solution: "Unnecessary" conversion
    func(&ref); // Takes pointer, restored clear communication, but needs the above conversion
}
```

4. Thank you for making this super valuable site. It's my first resource if I wanna look something up related to C++ :)

Last edited 5 months ago by SinixND

0 Reply

SinixND

Reply to SinixND 10 January 24, 2025 3:05 am PST

Addition to point 3 about pointer usage:
It might be useful to communicate side effects (in case one can not or does not want to avoid them).
In this case it might be useful to have a more or less cumbersome usage which might draw one away from modifying parameters without also returning them!

0 Reply

Almond_Blossom

January 10, 2025 10:47 am PST

I think I understand how in, out, and in-out parameters are classified, but I wanted to know if there's any sort of "rule of thumb" or method of remembering/understanding this more easily.

1 Reply

NordicPeace

December 17, 2024 9:06 pm PST

```
// Take Input as a reference of player level and give xp to player acc to that
int give_xp_points(const int& currentLevel, int& currentXP) {
    if (currentLevel < 10)  return currentXP += 50 ;
    else if (currentLevel == 10)  return currentXP += 100;
    else if (currentLevel == 15)  return currentXP += 500 ;
    else if (currentLevel == 20)  return currentXP += 1000 ;
    else return  currentXP += 2500 ;
}

int main() {
    int playerLevel{ 70 };
    int playerXp{ 20 };
    cout << give_xp_points(playerLevel, playerXp)<< endl;
}
```

Is this a correct use of out Parameters?

0 Reply

Alex Author

Reply to NordicPeace 11 December 28, 2024 7:23 pm PST

That's actually an in-out parameter, since the value passed in is used within the function (`currentXp += 100` is equivalent to `currentXp = currentXp + 100`). And yes, this works.

1 Reply

NordicCat

Reply to Alex 12 December 29, 2024 3:46 am PST

ohh! sorry for that. If you don't mind could you just explain me in Vs out Vs in-out in simple words actually it's hard for me to wrap my brain around this.

Last edited 6 months ago by NordicCat

0 Reply

Alex Author

Reply to NordicCat 13 January 3, 2025 8:03 pm PST

An out parameter is a parameter that is used only to pass a value back to the caller. Any value passed in by the caller is ignored. For example:

```
#include <iostream>

void foo(int& out)
{
    out = 5;
}

int main()
{
    int x { 1 };
    foo(x); // current value of 1 ignored by function, x changed
to 5
    std::cout << x << '\n'; // prints 5
}
```

An in-out parameter is just like an out parameter, except the value passed in is used somehow by the function.

```
#include <iostream>

void foo(int& inout)
{
    ++inout;
}

int main()
{
    int x { 1 };
    foo(x); // current value of 1 used by function, x changed to 2
    std::cout << x << '\n'; // prints 2
}
```

4 Reply

cpp learner
December 6, 2024 11:35 am PST

I had hard time to understand the point of this section even though I have some C++ dev experience.

I always *pass by reference *if I want to modify the variable.
If **const reference** is given, that means I won't change the variable

Same rule is applied for pointer arguments as well.
I also give const to the function to explicitly disallow member variable changes.

So in short, if I want to change a variable in a function, just passing by reference (IMO) all clear and always easy to understand

Is there a reason for passing by non const reference and not changing the variable that I am not aware of?

Last edited 6 months ago by cpp learner

0 Reply

Alex Author

Reply to cpp learner 14 December 9, 2024 9:07 pm PST

Nope. When using pass by reference, pass by non-const reference if you want to modify the argument, and pass by const reference otherwise.

Was there something that seemed to contradict this?

0 Reply

TEST

July 26, 2024 2:55 pm PDT

I don't know what happened here but the start of this lesson(12.13) was strong and then out of nowhere the quality fell dramatically after the "**Out parameters have an unnatural syntax**" section.

1 Reply

Alex Author

Reply to TEST 15 July 27, 2024 11:55 am PDT

Can you be more specific? This is too vague to be actionable. Thanks!

14 Reply

TEST

July 26, 2024 2:36 pm PDT

From section "**Out parameters**",

"We've named these out parameters with the suffix “out” to denote that they're out parameters. This helps remind the caller that the initial value passed to these parameters doesn't matter, and that we should expect them to be overwritten. By convention, output parameters are typically the rightmost parameters."

From section "**Out-parameters by reference don't make it obvious the arguments will be modified**",

"It is not clear from this function call that degrees is an in parameter, but sin and cos are out-parameters. If the caller does not realize that sin and cos will be modified, a semantic error will likely result."

These two statements are directly contradicting each other.

0 Reply

Alex Author

Reply to TEST 16 July 27, 2024 11:52 am PDT

No they aren't. The former is talking about the parameters in the function declaration, where the "out" suffix is useful. The latter is talking about the function call itself, where there is no easy way to differentiate an argument that is passed by value vs passed by reference.

5 Reply

siri
July 11, 2024 11:31 am PDT

What is Foo in this? a custom data type?

```
Foo someFcn(const Foo& in)
{
Foo foo { in }; // copy here
// modify foo
return foo;
}

int main()
{
Foo foo{};
foo = someFcn(foo); // makes it obvious foo is modified, but another copy made here

return 0;
}
```

Last edited 11 months ago by siri

0 Reply

Alex Author
Reply to siri 17 July 12, 2024 10:53 pm PDT

Yes, or a type alias.

0 Reply

Mello
July 1, 2024 10:24 pm PDT

Hey Alex!
I ran the sin/cos program in my compiler, and i am getting an interesting result for 90 degrees.
The sin comes out =1 just fine, but the cos is evaluated to 6.12323e-17
Why is this???
Same with 180 degrees cos= -1, but sin = 1.22465e-16

I know these values are very very close to zero, but mathematically should they not be an exact 0?

I have just completed high school with maths as one of my subjects, but idk what this is, is this some advanced level thing....or is there some error in code or how c++ evaluates sin and cos??

Last edited 1 year ago by Mello

0 Reply

Alex Author

Reply to Mello 18 July 4, 2024 2:05 pm PDT

Floating point numbers have limited precision. Our value of pi isn't exactly pi, and performing floating point arithmetic tends to accrue precision errors. So our radians calculation is slightly imprecise, and when std::sin or std::cos produces a result, it's also slightly imprecise.

This is an issue inherent to floating point numbers. We cover this in lesson https://www.learncpp.com/cpp-tutorial/floating-point-numbers/

1 Reply

Mello

Reply to Alex 19 July 5, 2024 12:41 am PDT

Oh, I see, floating point math is the problem, yeah that makes complete sense.
But hey, if our Pi is not exact pi, how come the values of 1 and -1 are evaluated precisely? Why does not the compiler round them to something like 0.99987999.... as it did with the value evaluating to zero?

Just my guess, but maybe values near zero are harder to handle, so it approximates them like this?

0 Reply

Alex Author

Reply to Mello 20 July 8, 2024 9:43 am PDT

Due to the way values are stored in IEEE754 format, a 64-bit IEEE754 double can precisely represent all integers with an absolute value of 2^53 or smaller.

The values that tend to get rounded are:

- Values that require an infinite recurring sequence of binary to represent.
- Values that have too many significant digits to store.

0 Reply

Mello

Reply to Alex 21 July 8, 2024 8:26 pm PDT

Okay, got it. Thanks!

0 Reply

MOEGMA25

June 15, 2024 6:46 am PDT

If you're going to pass by reference in order to avoid making a copy of the argument, you should almost always pass by const reference.

But doing this will still make you have to create a new object that will hold the const reference in order to make changes and return that changed value, which still results in a copy being made.

Is this copy of the reference considerd much more expensive than changing the reference?

Here is an example:

```cpp
#include <iostream>

int multiplier(const int& x) {
    int y{ x };
    y *= 2;
    return y;
}

int main()
{
    int x{ 5 };

    std::cout << multiplier(x);

    return 0;
}
```

Last edited 1 year ago by MOEGMA25

0 Reply

SinixND

Reply to MOEGMA25 22 January 24, 2025 1:20 am PST

What about returning an (technically unnecessary, but possibly enforced by [[nodiscard]]) const reference to communicate the modification of the inOut-Parameter without creating a copy. I benchmarked it, seems so have nearly no cost for bigger datatypes compared to making copies:

```cpp
[[nodiscard]]
Foo const& costlyModification(Foo& bigParameter)
{
    // modify big parameter here
    return bigParameter;
}

int main()
{
    Foo bigData{};
    bigData = costlyModification(bigData); // Communicates modification clearly
}
```

1 Reply

Alex Author

Reply to MOEGMA25 22 June 16, 2024 6:19 pm PDT

If the function is going to change the argument passed in, pass by non-const reference.

Pass by const reference is preferable in cases where we aren't modifying the argument (which is most cases).

3 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://stackoverflow.com/a/9779765
3. https://www.learncpp.com/cpp-tutorial/type-deduction-with-pointers-references-and-const/
4. https://www.learncpp.com/
5. https://www.learncpp.com/cpp-tutorial/return-by-reference-and-return-by-address/
6. https://www.learncpp.com/in-and-out-parameters/
7. https://www.learncpp.com/cpp-tutorial/pass-by-const-lvalue-reference/
8. https://www.learncpp.com/cpp-tutorial/introduction-to-destructors/
9. https://gravatar.com/
10. https://www.learncpp.com/cpp-tutorial/in-and-out-parameters/#comment-606913
11. https://www.learncpp.com/cpp-tutorial/in-and-out-parameters/#comment-605395
12. https://www.learncpp.com/cpp-tutorial/in-and-out-parameters/#comment-605854
13. https://www.learncpp.com/cpp-tutorial/in-and-out-parameters/#comment-605878
14. https://www.learncpp.com/cpp-tutorial/in-and-out-parameters/#comment-604976
15. https://www.learncpp.com/cpp-tutorial/in-and-out-parameters/#comment-600131
16. https://www.learncpp.com/cpp-tutorial/in-and-out-parameters/#comment-600124
17. https://www.learncpp.com/cpp-tutorial/in-and-out-parameters/#comment-599430
18. https://www.learncpp.com/cpp-tutorial/in-and-out-parameters/#comment-599116
19. https://www.learncpp.com/cpp-tutorial/in-and-out-parameters/#comment-599198
20. https://www.learncpp.com/cpp-tutorial/in-and-out-parameters/#comment-599207
21. https://www.learncpp.com/cpp-tutorial/in-and-out-parameters/#comment-599325
22. https://www.learncpp.com/cpp-tutorial/in-and-out-parameters/#comment-598450
23. https://g.ezoic.net/privacy/learncpp.com