12.13 — In and out parameters

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 address) 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:

```
#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] 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 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:

```
#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 of 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:

```
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:

```
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
    foo3(&i); // can modify i

    int *ptr { &i };
    foo3(ptr); // can modify i

    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 `foo(ptr)` allows `foo()` 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.

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.

```
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:

```
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:

```
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.

```
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.