11.3 — Passing arguments by reference

ALEX AUGUST 14, 2021

While pass by value is suitable in many cases, it has a couple of limitations. First, when passing a large struct or class to a function, pass by value will make a copy of the argument into the function parameter. In many cases, this is a needless performance hit, as the original argument would have sufficed. Second, when passing arguments by value, the only way to return a value back to the caller is via the function's return value. While this is often suitable, there are cases where it would be more clear and efficient to have the function modify the argument passed in. Pass by reference solves both of these issues.

Pass by reference

To pass a variable by reference, we simply declare the function parameters as references rather than as normal variables:

```
void addOne(int& ref) // ref is a reference variable
{
    ref = ref + 1;
}
```

When the function is called, ref will become a reference to the argument. Since a reference to a variable is treated exactly the same as the variable itself, any changes made to the reference are passed through to the argument!

The following example shows this in action:

```
void addOne(int& ref)
{
    ref = ref + 1;
}

int main()
{
    int value{ 5 };

    cout << "value = " << value << '\n';
    addOne(value);
    cout << "value = " << value << '\n';
    return 0;
}
```

This program is the same as the one we used for the pass by value example, except addOne's parameter is now a reference instead of a normal variable. When we call addOne(value), ref becomes a reference to main's value variable. This snippet produces the output:

```
value = 5
value = 6
```

As you can see, the function changed the value of the argument from 5 to 6!

Returning multiple values via out parameters

Sometimes we need a function to return multiple values. However, functions can only have one return value. One way to return multiple values is using reference parameters:

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

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 };

    // getSinCos will return the sin and cos in variables sin and cos
    getSinCos(30.0, sin, cos);

    std::cout << "The sin is " << sin << '\n';
    std::cout << "The cos is " << cos << '\n';
    return 0;
}
```

This function takes one parameter (by value) as input, and "returns" two parameters (by reference) as output. Parameters that are only used for returning values back to the caller are called **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 rewritten. 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, 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. 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.

This method, while functional, has a few minor downsides. First, the caller must pass in arguments to hold the updated outputs even if it doesn't intend to use them. More importantly, the syntax is a bit unnatural, with both the input and output parameters being put together in the function call. It's not obvious from the caller's end that sin and cos are out parameters and will be changed. This is probably the most dangerous part of this method (as it can lead to mistakes being made). Some programmers and companies feel this is a big enough problem

to advise avoiding output parameters altogether, or using pass by address for out parameters instead (which has a clearer syntax indicating whether a parameter is modifiable or not).

Personally, we recommend avoiding out parameters altogether if possible. If you do use them, naming out parameters (and output arguments) with an "out" suffix (or prefix) can help make it clear that the value might be modified.

Limitations of pass by reference

Non-const references can only reference non-const l-values (e.g. non-const variables), so a reference parameter cannot accept an argument that is a const l-value or an r-value (e.g. literals and the results of expressions).

Pass by const reference

As mentioned in the introduction, one of the major disadvantages of pass by value is that all arguments passed by value are copied into the function parameters. When the arguments are large structs or classes, this can take a lot of time. References provide a way to avoid this penalty. When an argument is passed by reference, a reference is created to the actual argument (which takes minimal time) and no copying of values takes place. This allows us to pass large structs and classes with a minimum performance penalty.

However, this also opens us up to potential trouble. References allow the function to change the value of the argument, which is undesirable when we want an argument be read-only. If we know that a function should not change the value of an argument, but don't want to pass by value, the best solution is to pass by const reference.

You already know that a const reference is a reference that does not allow the variable being referenced to be changed through the reference. Consequently, if we use a const reference as a parameter, we guarantee to the caller that the function will not change the argument!

The following function will produce a compiler error:

```
1 void foo(const std::string& x) // x is a const reference
{
    x = "hello";  // compile error: a const reference cannot have its value
changed!
}
```

Using const is useful for several reasons:

- It enlists the compilers help in ensuring values that shouldn't be changed aren't changed (the compiler will throw an error if you try, like in the above example).
- It tells the programmer that the function won't change the value of the argument. This can help with debugging.
- You can't pass a const argument to a non-const reference parameter. Using const parameters ensures you can pass both non-const and const arguments to the function.
- Const references can accept any type of argument, including non-const l-values, const l-values, and r-values.

Best practice

When passing an argument by reference, always use a const reference unless you need to change the value of the argument.

A reminder

Non-const references cannot bind to r-values. A function with a non-const reference parameter cannot be called with literals or temporaries.

```cpp
#include <string>

void foo(std::string& text) {}

int main()
{
  std::string text{ "hello" };

  foo(text); // ok
  foo(text + " world"); // illegal, non-const references can't bind to r-values.

  return 0;
}
```

References to pointers

It's possible to pass a pointer by reference, and have the function change the address of the pointer entirely:

```cpp
#include <iostream>

void foo(int*& ptr) // pass pointer by reference
{
 ptr = nullptr; // this changes the actual ptr argument passed in, not a copy
}

int main()
{
 int x{ 5 };
 int* ptr{ &x };
 std::cout << "ptr is: " << (ptr ? "non-null" : "null") << '\n'; // prints non-null
 foo(ptr);
 std::cout << "ptr is: " << (ptr ? "non-null" : "null") << '\n'; // prints null

 return 0;
}
```

(We'll show another example of this in the next lesson)

As a reminder, you can pass a C-style array by reference. This is useful if you need the ability for the function to change the array (e.g. for a sort function) or you need access to the array's type information of a fixed array (to do sizeof() or a for-each loop). However, note that in order for this to work, you explicitly need to define the array size in the parameter:

```cpp
#include <iostream>

// Note: You need to specify the array size in the function declaration
void printElements(int (&arr)[4])
{
  int length{ sizeof(arr) / sizeof(arr[0]) }; // we can now do this since the array won't decay

  for (int i{ 0 }; i < length; ++i)
  {
    std::cout << arr[i] << '\n';
  }
}

int main()
{
    int arr[]{ 99, 20, 14, 80 };

    printElements(arr);

    return 0;
}
```

This means this only works with fixed arrays of one particular length. If you want this to work with fixed arrays of any length, you can make the array length a template parameter (covered in a later chapter).

Pros and cons of pass by reference

Advantages of passing by reference:

- References allow a function to change the value of the argument, which is sometimes useful. Otherwise, const references can be used to guarantee the function won't change the argument.
- Because a copy of the argument is not made, pass by reference is fast, even when used with large structs or classes.
- References can be used to return multiple values from a function (via out parameters).
- References must be initialized, so there's no worry about null values.

Disadvantages of passing by reference:

- Because a non-const reference cannot be initialized with a const l-value or an r-value (e.g. a literal or an expression), arguments to non-const reference parameters must be normal variables.
- It can be hard to tell whether an argument passed by non-const reference is meant to be input, output, or both. Judicious use of const and a naming suffix for out variables can help.
- It's impossible to tell from the function call whether the argument may change. An argument passed by value and passed by reference looks the same. We can only tell whether an argument is passed by value or reference by looking at the function declaration. This can lead to situations where the programmer does not realize a function will change the value of the argument.

When to use pass by reference:

- When passing structs or classes (use const if read-only).
- When you need the function to modify an argument.
- When you need access to the type information of a fixed array.

When not to use pass by reference:

- When passing fundamental types that don't need to be modified (use pass by value).

Best practice

Use pass by (const) reference instead of pass by value for structs and classes and other expensive-to-copy types.

Next lesson

11.4 Passing arguments by address

Back to table of contents

Previous lesson

11.2 Passing arguments by value

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

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

Notify me about replies: POST COMMENT

DP N N FOUT

Newest