11.4 — Passing arguments by address

ALEX JULY 13, 2021

There is one more way to pass variables to functions, and that is by address. Passing an argument by address involves passing the address of the argument variable rather than the argument variable itself. Because the argument is an address, the function parameter must be a pointer. The function can then dereference the pointer to access or change the value being pointed to.

Here is an example of a function that takes a parameter passed by address:

```cpp
#include <iostream>

void foo(int* ptr)
{
    *ptr = 6;
}

int main()
{
    int value{ 5 };

    std::cout << "value = " << value << '\n';
    foo(&value);
    std::cout << "value = " << value << '\n';
    return 0;
}
```

The above snippet prints:

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

As you can see, the function foo() changed the value of the argument (variable `value`) through pointer parameter ptr.

Pass by address is typically used with pointers, which most often are used to point to built-in arrays. For example, the following function will print all the values in an array:

```cpp
void printArray(int* array, int length)
{
    for (int index{ 0 }; index < length; ++index)
    {
        std::cout << array[index] << ' ';
    }
}
```

Here is an example program that calls this function:

```cpp
int main()
{
    int array[6]{ 6, 5, 4, 3, 2, 1 }; // remember, arrays decay into pointers
    printArray(array, 6); // so array evaluates to a pointer to the first element of the array here, no & needed
}
```

This program prints the following:

```
6 5 4 3 2 1
```

Remember that fixed arrays decay into pointers when passed to a function, so we have to pass the length as a separate parameter.

It is always a good idea to ensure parameters passed by address are not null pointers before dereferencing them. Dereferencing a null pointer will typically cause the program to crash. Here is our printArray() function with a null pointer check:

```cpp
void printArray(int* array, int length)
{
    // if user passed in a null pointer for array, bail out early!
    if (!array)
        return;

    for (int index{ 0 }; index < length; ++index)
        std::cout << array[index] << ' ';
}

int main()
{
    int array[6]{ 6, 5, 4, 3, 2, 1 };
    printArray(array, 6);
}
```

Passing by const address

Because printArray() doesn't modify any of its arguments, it's good form to make the array parameter const:

```cpp
void printArray(const int* array, int length)
{
    // if user passed in a null pointer for array, bail out early!
    if (!array)
        return;

    for (int index{ 0 }; index < length; ++index)
        std::cout << array[index] << ' ';
}

int main()
{
    int array[6]{ 6, 5, 4, 3, 2, 1 };
    printArray(array, 6);
}
```

This allows us to tell at a glance that printArray() won't modify the array argument passed in, and will ensure we don't do so by accident.

Addresses are actually passed by value

When you pass a pointer to a function, the pointer's value (the address it points to) is copied from the argument to the function's parameter. In other words, it's passed by value! If you change the function parameter's value, you are only changing a copy. Consequently, the original pointer argument will not be changed.

Here's a sample program that illustrates this.

```cpp
#include <iostream>

void setToNull(int* tempPtr)
{
    // we're making tempPtr point at something else, not changing the value that tempPtr points to.
    tempPtr = nullptr; // use 0 instead if not C++11
}

int main()
{
    // First we set ptr to the address of five, which means *ptr = 5
    int five{ 5 };
    int* ptr{ &five };

    // This will print 5
    std::cout << *ptr;

    // tempPtr will receive a copy of ptr
    setToNull(ptr);

    // ptr is still set to the address of five!

    // This will print 5
    if (ptr)
        std::cout << *ptr;
    else
        std::cout << " ptr is null";

    return 0;
}
```

tempPtr receives a copy of the address that ptr is holding. Even though we change tempPtr to point at something else (nullptr), this does not change the value that ptr points to. Consequently, this program prints:

```
55
```

Note that even though the address itself is passed by value, you can still dereference that address to change the argument's value. This is a common point of confusion, so let's clarify:

- When passing an argument by address, the function parameter variable receives a copy of the address from the argument. At this point, the function parameter and the argument both point to the same value.
- If the function parameter is then *dereferenced* to change the value being pointed to, that *will* impact the value the argument is pointing to, since both the function parameter and argument are pointing to the same value!
- If the function parameter is *assigned* a different address, that will not impact the argument, since the function parameter is a copy, and changing the copy won't impact the original. After changing the function parameter's address, the function parameter and argument will point to different values, so dereferencing the parameter and changing the value will no longer affect the value pointed to by the argument.

The following program illustrates the point:

```cpp
#include <iostream>

void setToSix(int* tempPtr)
{
    *tempPtr = 6; // we're changing the value that tempPtr (and ptr) points to
}

int main()
{
    // First we set ptr to the address of five, which means *ptr = 5
    int five{ 5 };
    int* ptr{ &five };

    // This will print 5
    std::cout << *ptr;

    // tempPtr will receive a copy of ptr
    setToSix(ptr);

    // tempPtr changed the value being pointed to to 6, so ptr is now pointing to the value 6

    // This will print 6
    if (ptr)
        std::cout << *ptr;
    else
        std::cout << " ptr is null";

    return 0;
}
```

This prints:

```
56
```

Passing addresses by reference

The next logical question is, "What if we want to change the address an argument points to from within the function?". Turns out, this is surprisingly easy. You can simply pass the address by reference. The syntax for doing a reference to a pointer is a little strange (and easy to get backwards). However, if you do get it backwards, the compiler will give you an error.

The following program illustrates using a reference to a pointer:

```cpp
#include <iostream>

// tempPtr is now a reference to a pointer, so any changes made to tempPtr will change the argument as well!
void setToNull(int*& tempPtr)
{
    tempPtr = nullptr; // use 0 instead if not C++11
}

int main()
{
    // First we set ptr to the address of five, which means *ptr = 5
    int five{ 5 };
    int* ptr{ &five };

    // This will print 5
    std::cout << *ptr;

    // tempPtr is set as a reference to ptr
    setToNull(ptr);

    // ptr has now been changed to nullptr!

    if (ptr)
        std::cout << *ptr;
    else
        std::cout << " ptr is null";

    return 0;
}
```

When we run the program again with this version of the function, we get:

```
5 ptr is null
```

Which shows that calling setToNull() did indeed change the value of ptr from &five to nullptr!

There is only pass by value

Now that you understand the basic differences between passing by reference, address, and value, let’s get reductionist for a moment. :)

In the lesson 10.16 -- Reference variables, we briefly mentioned that references are typically implemented by the compiler as pointers. This means that behind the scenes, pass by reference is essentially just a pass by address (with access to the reference doing an implicit dereference).

And just above, we showed that pass by address is actually just passing an address by value!

Therefore, we can conclude that C++ really passes everything by value! The properties of pass by address (and reference) come *solely* from the fact that we can dereference the passed address to change the argument, which we can not do with a normal value parameter!

Pass by address makes modifiable parameters explicit

Consider the following example:

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

int i {};

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

It’s not obvious from the call to foo2() that the function can modify variable i, is it?

For this reason, some guides recommend passing all modifiable arguments by address, so that it’s more obvious from an existing function call that an argument could be modified.

However, this comes with its own set of downsides: the caller might think they can pass in nullptr when they aren’t supposed to, and you now have to rigorously check for null pointers.

We lean towards the recommendation of passing non-optional modifiable parameters by reference. Even better, avoid modifiable parameters altogether.

Pros and cons of pass by address

Advantages of passing by address:

- Pass by address allows a function to change the value of the argument, which is sometimes useful. Otherwise, const can be used to guarantee the function won't change the argument. (However, if you want to do this with a non-pointer, you should use pass by reference instead).
- Because a copy of the argument is not made, it is fast, even when used with large structs or classes.
- We can return multiple values from a function via out parameters.

Disadvantages of passing by address:

- Because literals (excepting C-style string literals) and expressions do not have addresses, pointer arguments must be normal variables.
- All values must be checked to see whether they are null. Trying to dereference a null value will result in a crash. It is easy to forget to do this.
- Because dereferencing a pointer is slower than accessing a value directly, accessing arguments passed by address is slower than accessing arguments passed by value.

When to use pass by address:

- When passing built-in arrays (if you're okay with the fact that they'll decay into a pointer).
- When passing a pointer and nullptr is a valid argument logically.

When not to use pass by address:

- When passing a pointer and nullptr is not a valid argument logically (use pass by reference).
- When passing structs or classes (use pass by reference).
- When passing fundamental types (use pass by value).

As you can see, pass by address and pass by reference have almost identical advantages and disadvantages. Because pass by reference is generally safer than pass by address, pass by reference should be preferred in most cases.

Best practice

Prefer pass by reference to pass by address whenever applicable.

Next lesson
11.5 Returning values by value, reference, and address

Back to table of contents

Previous lesson
11.3 Passing arguments by reference

Newest

X

X