12.11 — Pass by address (part 2)

ALEX SEPTEMBER 11, 2023

This lesson is a continuation of 12.10 -- Pass by address (https://www.learncpp.com/cpp-tutorial/pass-by-address/).

Pass by address for "optional" arguments

One of the more common uses for pass by address is to allow a function to accept an "optional" argument. This is easier to illustrate by example than to describe:

```
#include <iostream>
#include <string>

void greet(std::string* name=nullptr)
{
    std::cout << "Hello ";
    std::cout << (name ? *name : "guest") << '\n';
}

int main()
{
    greet(); // we don't know who the user is yet

    std::string joe{ "Joe" };
    greet(&joe); // we know the user is joe

    return 0;
}
```

This example prints:

```
Hello guest
Hello Joe
```

In this program, the `greet()` function has one parameter that is passed by address and defaulted to `nullptr`. Inside `main()`, we call this function twice. The first call, we don't know who the user is, so we call `greet()` without an argument. The `name` parameter defaults to `nullptr`, and the greet function substitutes in the name "guest". For the second call, we now have a valid user, so we call `greet(&joe)`. The `name` parameter receives the address of `joe`, and can use it to print the name "Joe".

However, in many cases, function overloading is a better alternative to achieve the same result:

```
#include <iostream>
#include <string>
#include <string_view>

void greet(std::string_view name)
{
    std::cout << "Hello " << name << '\n';
}

void greet()
{
    greet("guest");
}

int main()
{
    greet(); // we don't know who the user is yet

    std::string joe{ "Joe" };
    greet(joe); // we know the user is joe

    return 0;
}
```

This has a number of advantages: we no longer have to worry about null dereferences, and we could pass in a string literal if we wanted.

Changing what a pointer parameter points at

When we pass an address to a function, that address is copied from the argument into the pointer parameter (which is fine, because copying an address is fast). Now consider the following program:

```
#include <iostream>

// [[maybe_unused]] gets rid of compiler warnings about ptr2 being set but not used
void nullify([[maybe_unused]] int* ptr2)
{
    ptr2 = nullptr; // Make the function parameter a null pointer
}

int main()
{
    int x{ 5 };
    int* ptr{ &x }; // ptr points to x

    std::cout << "ptr is " << (ptr ? "non-null\n" : "null\n");

    nullify(ptr);

    std::cout << "ptr is " << (ptr ? "non-null\n" : "null\n");
    return 0;
}
```

This program prints:

```
ptr is non-null
ptr is non-null
```

As you can see, changing the address held by the pointer parameter had no impact on the address held by the argument (`ptr` still points at `x`). When function `nullify()` is called, `ptr2` receives a copy of the address passed in (in this case, the address held by `ptr`, which is the address of `x`). When the function changes what `ptr2` points at, this only affects the copy held by `ptr2`.

So what if we want to allow a function to change what a pointer argument points to?

Pass by address... by reference?

Yup, it's a thing. Just like we can pass a normal variable by reference, we can also pass pointers by reference. Here's the same program as above with `ptr2` changed to be a reference to an address:

```
#include <iostream>

void nullify(int*& refptr) // refptr is now a reference to a pointer
{
    refptr = nullptr; // Make the function parameter a null pointer
}

int main()
{
    int x{ 5 };
    int* ptr{ &x }; // ptr points to x

    std::cout << "ptr is " << (ptr ? "non-null\n" : "null\n");

    nullify(ptr);

    std::cout << "ptr is " << (ptr ? "non-null\n" : "null\n");
    return 0;
}
```

This program prints:

```
ptr is non-null
ptr is null
```

Because `refptr` is now a reference to a pointer, when `ptr` is passed as an argument, `refptr` is bound to `ptr`. This means any changes to `refptr` are made to `ptr`.

As an aside...

Because references to pointers are fairly uncommon, it can be easy to mix up the syntax (is it `int*&` or `int&*`?). The good news is that if you do it backwards, the compiler will error because you can't have a pointer to a reference (because pointers must hold the address of an object, and references aren't objects). Then you can switch it around.

Why using `0` or `NULL` is no longer preferred (optional)

In this subsection, we'll explain why using `0` or `NULL` is no longer preferred.

The literal `0` can be interpreted as either an integer literal, or as a null pointer literal. In certain cases, it can be ambiguous which one we intend -- and in some of those cases, the compiler may assume we mean one when we mean the other -- with unintended consequences to the behavior of our program.

The definition of preprocessor macro `NULL` is not defined by the language standard. It can be defined as `0`, `0L`, `((void*)0)`, or something else entirely.

In lesson 11.1 -- Introduction to function overloading (https://www.learncpp.com/cpp-tutorial/introduction-to-function-overloading/), we discussed that functions can be overloaded (multiple functions can have the same name, so long as they can be differentiated by the number or type of parameters). The compiler can figure out which overloaded function you desire by the arguments passed in as part of the function call.

When using `0` or `NULL`, this can cause problems:

```
#include <iostream>
#include <cstddef> // for NULL

void print(int x) // this function accepts an integer
{
    std::cout << "print(int): " << x << '\n';
}

void print(int* ptr) // this function accepts an integer pointer
{
    std::cout << "print(int*): " << (ptr ? "non-null\n" : "null\n");
}

int main()
{
    int x{ 5 };
    int* ptr{ &x };

    print(ptr);  // always calls print(int*) because ptr has type int* (good)
    print(0);    // always calls print(int) because 0 is an integer literal (hopefully this is what we expected)

    print(NULL); // this statement could do any of the following:
    // call print(int) (Visual Studio does this)
    // call print(int*)
    // result in an ambiguous function call compilation error (gcc and Clang do this)

    print(nullptr); // always calls print(int*)

    return 0;
}
```

On the author's machine (using Visual Studio), this prints:

```
print(int*): non-null
print(int): 0
print(int): 0
print(int*): null
```

When passing integer value `0` as a parameter, the compiler will prefer `print(int)` over `print(int*)`. This can lead to unexpected results when we intended `print(int*)` to be called with a null pointer argument.

In the case where `NULL` is defined as value `0`, `print(NULL)` will also call `print(int)`, not `print(int*)` like you might expect for a null pointer literal. In cases where `NULL` is not defined as `0`, other behavior might result, like a call to `print(int*)` or a compilation error.

Using `nullptr` removes this ambiguity (it will always call `print(int*)`), since `nullptr` will only match a pointer type.

std::nullptr_t (optional)

Since `nullptr` can be differentiated from integer values in function overloads, it must have a different type. So what type is `nullptr`? The answer is that `nullptr` has type `std::nullptr_t` (defined in header <cstddef>). `std::nullptr_t` can only hold one value: `nullptr`! While this may seem kind of silly, it's useful in one situation. If we want to write a function that accepts only a `nullptr` literal argument, we can make the parameter a `std::nullptr_t`.

```cpp
#include <iostream>
#include <cstddef> // for std::nullptr_t

void print(std::nullptr_t)
{
    std::cout << "in print(std::nullptr_t)\n";
}

void print(int*)
{
    std::cout << "in print(int*)\n";
}

int main()
{
    print(nullptr); // calls print(std::nullptr_t)

    int x { 5 };
    int* ptr { &x };

    print(ptr); // calls print(int*)

    ptr = nullptr;
    print(ptr); // calls print(int*) (since ptr has type int*)

    return 0;
}
```

In the above example, the function call `print(nullptr)` resolves to the function `print(std::nullptr_t)` over `print(int*)` because it doesn't require a conversion.

The one case that might be a little confusing is when we call `print(ptr)` when `ptr` is holding the value `nullptr`. Remember that function overloading matches on types, not values, and `ptr` has type `int*`. Therefore, `print(int*)` will be matched. `print(std::nullptr_t)` isn't even in consideration in this case, as pointer types will not implicitly convert to a `std::nullptr_t`.

You probably won't ever need to use this, but it's good to know, just in case.

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. :)

While the compiler can often optimize references away entirely, there are cases where this is not possible and a reference is actually needed. References are normally implemented by the compiler using 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 in the previous lesson, we mentioned that pass by address just copies an address from the caller to the called function -- which is 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!

Next lesson
12.12 Return by reference and return by address

Back to table of contents

Previous lesson
12.10 Pass by address