LEARN C++
Skill up with our free tutorials

12.11 — Pass by address (part 2)

ALEX[1] APRIL 25, 2024

This lesson is a continuation of 12.10 -- Pass by address (https://www.learncpp.com/cpp-tutorial/pass-by-address/)[2].

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:

```cpp
#include <iostream>

void printIDNumber(const int *id=nullptr)
{
    if (id)
        std::cout << "Your ID number is " << *id << ".\n";
    else
        std::cout << "Your ID number is not known.\n";
}

int main()
{
    printIDNumber(); // we don't know the user's ID yet

    int userid { 34 };
    printIDNumber(&userid); // we know the user's ID now

    return 0;
}
```

This example prints:

```
Your ID number is not known.
Your ID number is 34.
```

In this program, the `printIDNumber()` 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 the user’s ID, so we call `printIDNumber()` without an argument. The `id` parameter defaults to `nullptr`, and the function prints `Your ID number is not known.`. For the second call, we now have a valid id, so we call `printIDNumber(&userid)`. The `id` parameter receives the address of `userid`, so the function prints `Your ID number is 34.`.

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

```
#include <iostream>

void printIDNumber()
{
    std::cout << "Your ID is not known\n";
}

void printIDNumber(int id)
{
    std::cout << "Your ID is " << id << "\n";
}

int main()
{
    printIDNumber(); // we don't know the user's ID yet

    int userid { 34 };
    printIDNumber(userid); // we know the user is 34

    printIDNumber(62); // now also works with rvalue arguments

    return 0;
}
```

This has a number of advantages: we no longer have to worry about null dereferences, and we can pass in literals or other rvalues as an argument.

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:

```cpp
#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/)[3], 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.

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

4

Back to table of contents

5

Previous lesson
12.10 Pass by address

2

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.

119 COMMENTS Newest

Tobito
March 1, 2025 2:48 am PST

Pass by value (copy value to parameter)
Pass by address (copy address to parameter)

Pass by reference? It only bound to argument? No any copy to parameter? It's right?

1 Reply

Dat

February 21, 2025 1:08 am PST

The more i read about c++ it did not confuse me it just a whole new level with extra stuff.

1 Reply

nurn

January 20, 2025 1:39 am PST

Feels like were picking up some steam around here..

4 Reply

KLAP

October 21, 2024 2:02 pm PDT

From this chapter what I can reduce to is a decent amount of mistakes were commonly made passing addresses around using pointers, so reference was born. Or was it always there?

0 Reply

> Alex Author
>
> Reply to KLAP 10 October 21, 2024 3:07 pm PDT
>
> C doesn't have references. They were added in C++ as a way to allow functions to pass (and possibly change) arguments without making a copy, in a safer way.
>
> 7 Reply

Bili

October 17, 2024 9:38 am PDT

in example of "Changing what a pointer parameter points at", what do u think about using pointer to pointer?. i tried this code and it works :

```
#include <iostream>

void nullify(int **ptr){
    *ptr = nullptr;
}

int main(){
    int num = 5;
    int* num_ptr = &num;

    std::cout<<"num_ptr is"<<(num_ptr? "non-null" : "null")<<std::endl;

    nullify(&num_ptr);
    std::cout<<"num_ptr is "<<(num_ptr? "non-null" : "null")<<std::endl;

    num_ptr = nullptr;
    std::cout<<"num_ptr is "<<(num_ptr? "non-null" : "null")<<std::endl;

    return 0;
}
```

i admit it's confusing, but any other reason why u dont use this?

0 Reply

Alex Author

Reply to Bili 11 October 19, 2024 6:12 pm PDT

Because it's confusing. :) Using a reference is slightly clearer.

0 Reply

redshift

Reply to Alex 12 May 4, 2025 3:35 pm PDT

It's not confusing to me. It's clear you're passing the address of the pointer and then changing the value that pointer holds to nullptr.

The reference is more confusing to me, so it's personal preference.

0 Reply

Corvin

August 20, 2024 5:02 am PDT

```
void printIDNumber(int id)
{
    std::cout << "Your ID is " << id << "\n";
}
```

why is `printIDNumber(int id)` not `printIDNumber(const int& ref id)`?

0 Reply

Alex Author

Reply to Corvin [13] August 20, 2024 9:07 pm PDT

Because it's preferable to pass fundamental types by value. We discuss this in lesson https://www.learncpp.com/cpp-tutorial/pass-by-const-lvalue-reference/

👍 0 Reply

Spesader

May 16, 2024 10:16 am PDT

I like the plot twist at the end :D

👍 36 Reply

NordicPeace

Reply to Spesader [14] December 17, 2024 9:06 am PST

i know it was always our hero reference that could save us from the Monster Pointer.

👍 2 Reply

Asicx

April 22, 2024 5:33 pm PDT

"(with access to the reference doing an implicit dereference)."
with access to the reference "by" doing an implicit dereference.
Can be a little confusing so i pointed it out.

Last edited 1 year ago by Asicx

👍 0 Reply

Alex Author

Reply to Asicx [15] April 25, 2024 2:03 pm PDT

I removed the sentence. While interesting, it isn't required for the point I'm trying to make.

👍 0 Reply

Asicx

Reply to Alex [16] April 25, 2024 5:25 pm PDT

I understand. Knowing how references are implemented by the compiler (how they work behind the scene) was very interesting to me because i didn't fully understand the difference between the two (pointer and reference).
Someone asked the same question i had in mind on stackoverflow (https://stackoverflow.com/questions/3954764/how-are-references-implemented-internally)[17]
Interesting indeed.

0 Reply

Phargelm
April 9, 2024 5:25 am PDT

> However, in many cases, function overloading is a better alternative to achieve the same result: <code example> 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.

These advantages are not the result of using overloading instead of default parameters, it's a result of replacing pointer with `std::string_view`. We can achieve the same advantages using default parameters as well:

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

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

int main()
{
    greet();

    std::string joe{ "Joe" };
    greet(joe);

    return 0;
}
```

Last edited 1 year ago by Phargelm

0 Reply

Alex Author
Reply to Phargelm 18 April 10, 2024 5:11 pm PDT

Thanks for the feedback. I rewrote the examples to better reflect what I'm trying to describe.

0 Reply

Phargelm
Reply to Alex 19 April 11, 2024 6:44 am PDT

Perhaps it worth to change the related example in std::optional lesson (section "Passing a std::optional") as it also uses `void greet(std::optional<std::string> name=std::nullopt)` that can be substituted with void `greet(std::string_view name="guest")`
https://www.learncpp.com/cpp-tutorial/stdoptional/

0 Reply

Alex Author

Reply to Phargelm[20] April 13, 2024 9:13 pm PDT

Updated the std::optional lesson as well. Thanks for pointing this out.

0 Reply

AnIdiotIndeed

April 4, 2024 7:01 pm PDT

So finally pass by reference is just pass by address(with an implicit dereference) which is just a pass by value of an address of an l-value...right?

6 Reply

Alex Author

Reply to AnIdiotIndeed[21] April 5, 2024 9:15 am PDT

Yep, assuming the reference isn't optimized out.

3 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/pass-by-address/
3. https://www.learncpp.com/cpp-tutorial/introduction-to-function-overloading/
4. https://www.learncpp.com/cpp-tutorial/return-by-reference-and-return-by-address/
5. https://www.learncpp.com/
6. https://www.learncpp.com/pass-by-address-part-2/
7. https://www.learncpp.com/cpp-tutorial/value-categories-lvalues-and-rvalues/
8. https://www.learncpp.com/cpp-tutorial/introduction-to-program-defined-user-defined-types/
9. https://gravatar.com/
10. https://www.learncpp.com/cpp-tutorial/pass-by-address-part-2/#comment-603422
11. https://www.learncpp.com/cpp-tutorial/pass-by-address-part-2/#comment-603235
12. https://www.learncpp.com/cpp-tutorial/pass-by-address-part-2/#comment-603358
13. https://www.learncpp.com/cpp-tutorial/pass-by-address-part-2/#comment-601056
14. https://www.learncpp.com/cpp-tutorial/pass-by-address-part-2/#comment-597162
15. https://www.learncpp.com/cpp-tutorial/pass-by-address-part-2/#comment-596098
16. https://www.learncpp.com/cpp-tutorial/pass-by-address-part-2/#comment-596203
17. https://stackoverflow.com/questions/3954764/how-are-references-implemented-internally
18. https://www.learncpp.com/cpp-tutorial/pass-by-address-part-2/#comment-595584
19. https://www.learncpp.com/cpp-tutorial/pass-by-address-part-2/#comment-595641

20. https://www.learncpp.com/cpp-tutorial/pass-by-address-part-2/#comment-595655
21. https://www.learncpp.com/cpp-tutorial/pass-by-address-part-2/#comment-595450
22. https://g.ezoic.net/privacy/learncpp.com