12.10 — Pass by address

ALEX[1] FEBRUARY 13, 2025

In prior lessons, we've covered two different ways to pass an argument to a function: pass by value (2.4 -- Introduction to function parameters and arguments (https://www.learncpp.com/cpp-tutorial/introduction-to-function-parameters-and-arguments/)[2]) and pass by reference (12.5 -- Pass by lvalue reference (https://www.learncpp.com/cpp-tutorial/pass-by-lvalue-reference/)[3]).

Here's a sample program that shows a `std::string` object being passed by value and by reference:

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

void printByValue(std::string val) // The function parameter is a copy of str
{
    std::cout << val << '\n'; // print the value via the copy
}

void printByReference(const std::string& ref) // The function parameter is a reference that binds to str
{
    std::cout << ref << '\n'; // print the value via the reference
}

int main()
{
    std::string str{ "Hello, world!" };

    printByValue(str); // pass str by value, makes a copy of str
    printByReference(str); // pass str by reference, does not make a copy of str

    return 0;
}
```

When we pass argument `str` by value, the function parameter `val` receives a copy of the argument. Because the parameter is a copy of the argument, any changes to the `val` are made to the copy, not the original argument.

When we pass argument `str` by reference, the reference parameter `ref` is bound to the actual argument. This avoids making a copy of the argument. Because our reference parameter is const, we are not allowed to change `ref`. But if `ref` were non-const, any changes we made to `ref` would change `str`.

In both cases, the caller is providing the actual object (`str`) to be passed as an argument to the function call.

Pass by address

C++ provides a third way to pass values to a function, called pass by address. With **pass by address**, instead of providing an object as an argument, the caller provides an object's *address* (via a pointer). This pointer (holding the address of the object) is copied into a pointer parameter of the called function (which now also holds the address of the object). The function can then dereference that pointer to access the object whose address was passed.

Here’s a version of the above program that adds a pass by address variant:

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

void printByValue(std::string val) // The function parameter is a copy of str
{
    std::cout << val << '\n'; // print the value via the copy
}

void printByReference(const std::string& ref) // The function parameter is a reference that binds to str
{
    std::cout << ref << '\n'; // print the value via the reference
}

void printByAddress(const std::string* ptr) // The function parameter is a pointer that holds the address of str
{
    std::cout << *ptr << '\n'; // print the value via the dereferenced pointer
}

int main()
{
    std::string str{ "Hello, world!" };

    printByValue(str); // pass str by value, makes a copy of str
    printByReference(str); // pass str by reference, does not make a copy of str
    printByAddress(&str); // pass str by address, does not make a copy of str

    return 0;
}
```

Note how similar all three of these versions are. Let’s explore the pass by address version in more detail.

First, because we want our `printByAddress()` function to use pass by address, we’ve made our function parameter a pointer named `ptr`. Since `printByAddress()` will use `ptr` in a read-only manner, `ptr` is a pointer to a const value.

```cpp
void printByAddress(const std::string* ptr)
{
    std::cout << *ptr << '\n'; // print the value via the dereferenced pointer
}
```

Inside the `printByAddress()` function, we dereference `ptr` parameter to access the value of the object being pointed to.

Second, when the function is called, we can’t just pass in the `str` object -- we need to pass in the address of `str`. The easiest way to do that is to use the address-of operator (&) to get a pointer holding the address of `str`:

```cpp
printByAddress(&str); // use address-of operator (&) to get pointer holding address of str
```

When this call is executed, `&str` will create a pointer holding the address of `str`. This address is then copied into function parameter `ptr` as part of the function call. Because `ptr` now holds the address of `str`, when the function dereferences `ptr`, it will get the value of `str`, which the function prints to the console.

That's it.

Although we use the address-of operator in the above example to get the address of `str`, if we already had a pointer variable holding the address of `str`, we could use that instead:

```
int main()
{
    std::string str{ "Hello, world!" };

    printByValue(str); // pass str by value, makes a copy of str
    printByReference(str); // pass str by reference, does not make a copy of str
    printByAddress(&str); // pass str by address, does not make a copy of str

    std::string* ptr { &str }; // define a pointer variable holding the address of str
    printByAddress(ptr); // pass str by address, does not make a copy of str

    return 0;
}
```

Nomenclature

When we pass the address of a variable as an argument using `operator&`, we say the variable is passed by address.

When we have a pointer variable holding the address of an object, and we pass the pointer as an argument to a parameter of the same type, we say the object is passed by address, and the pointer is passed by value.

Pass by address does not make a copy of the object being pointed to

Consider the following statements:

```
std::string str{ "Hello, world!" };
printByAddress(&str); // use address-of operator (&) to get pointer holding address of
str
```

As we noted in [12.5 -- Pass by lvalue reference](https://www.learncpp.com/cpp-tutorial/pass-by-lvalue-reference/) (https://www.learncpp.com/cpp-tutorial/pass-by-lvalue-reference/)[3], copying a `std::string` is expensive, so that's something we want to avoid. When we pass a `std::string` by address, we're not copying the actual `std::string` object -- we're just copying the pointer (holding the address of the object) from the caller to the called function. Since an address is typically only 4 or 8 bytes, a pointer is only 4 or 8 bytes, so copying a pointer is always fast.

Thus, just like pass by reference, pass by address is fast, and avoids making a copy of the argument object.

Pass by address allows the function to modify the argument's value

When we pass an object by address, the function receives the address of the passed object, which it can access via dereferencing. Because this is the address of the actual argument object being passed (not a copy of the object), if the function parameter is a pointer to non-const, the function can modify the argument via the pointer parameter:

```cpp
#include <iostream>

void changeValue(int* ptr) // note: ptr is a pointer to non-const in this example
{
    *ptr = 6; // change the value to 6
}

int main()
{
    int x{ 5 };

    std::cout << "x = " << x << '\n';

    changeValue(&x); // we're passing the address of x to the function

    std::cout << "x = " << x << '\n';

    return 0;
}
```

This prints:

```
x = 5
x = 6
```

As you can see, the argument is modified and this modification persists even after `changeValue()` has finished running.

If a function is not supposed to modify the object being passed in, the function parameter should be made a pointer-to-const:

```cpp
void changeValue(const int* ptr) // note: ptr is now a pointer to a const
{
    *ptr = 6; // error: can not change const value
}
```

For many of the same reasons we typically don't `const` regular (non-pointer, non-reference) function parameters (discussed in 5.1 -- Constant variables (named constants) (https://www.learncpp.com/cpp-tutorial/constant-variables-named-constants/)[4]), we also typically don't `const` pointer function parameters. Let's make two assertions:

- A `const` keyword used to make a pointer function parameter a const pointer provides little value (since it has no impact on the caller, and mostly serves as documentation that the pointer won't change).
- A `const` keyword used to differentiate a pointer-to-const from a pointer-to-non-const that can modify the object passed in is significant (as the caller needs to know if the function could change the value of the argument).

If we only use non-const pointer function parameters, then all uses of `const` are significant. As soon as we start using `const` for const pointer function parameters, then it becomes more difficult to determine whether a given use of `const` is significant or not. More importantly, it also makes it harder to notice pointer-to-non-const parameters. For example:

```
void foo(const char* source, char* dest, int count);             // Using non-const pointers, all consts are significant.
void foo(const char* const source, char* const dest, int count); // Using const pointers, `dest` being a pointer-to-non-const may go unnoticed amongst the sea of spurious consts.
```

In the former case, it’s easy to see that `source` is a pointer-to-const and `dest` is a pointer-to-non-const. In the latter case, it’s much harder to see that `dest` is a const-pointer-to-non-const, whose pointed-to object can be modified by the function!

Best practice

Prefer pointer-to-const function parameters over pointer-to-non-const function parameters, unless the function needs to modify the object passed in.
Do not make function parameters const pointers unless there is some specific reason to do so.

Null checking

Now consider this fairly innocent looking program:

```
#include <iostream>

void print(int* ptr)
{
    std::cout << *ptr << '\n';
}

int main()
{
    int x{ 5 };
    print(&x);

    int* myPtr {};
    print(myPtr);

    return 0;
}
```

When this program is run, it will print the value `5` and then most likely crash.

In the call to `print(myPtr)`, `myPtr` is a null pointer, so function parameter `ptr` will also be a null pointer. When this null pointer is dereferenced in the body of the function, undefined behavior results.

When passing a parameter by address, care should be taken to ensure the pointer is not a null pointer before you dereference the value. One way to do that is to use a conditional statement:

```cpp
#include <iostream>

void print(int* ptr)
{
    if (ptr) // if ptr is not a null pointer
    {
        std::cout << *ptr << '\n';
    }
}

int main()
{
    int x{ 5 };

    print(&x);
    print(nullptr);

    return 0;
}
```

In the above program, we're testing `ptr` to ensure it is not null before we dereference it. While this is fine for such a simple function, in more complicated functions this can result in redundant logic (testing if ptr is not null multiple times) or nesting of the primary logic of the function (if contained in a block).

In most cases, it is more effective to do the opposite: test whether the function parameter is null as a precondition (9.6 -- Assert and static_assert (https://www.learncpp.com/cpp-tutorial/assert-and-static_assert/)[5]) and handle the negative case immediately:

```cpp
#include <iostream>

void print(int* ptr)
{
    if (!ptr) // if ptr is a null pointer, early return back to the caller
        return;

    // if we reached this point, we can assume ptr is valid
    // so no more testing or nesting required

    std::cout << *ptr << '\n';
}

int main()
{
    int x{ 5 };

    print(&x);
    print(nullptr);

    return 0;
}
```

If a null pointer should never be passed to the function, an `assert` (which we covered in lesson 9.6 -- Assert and static_assert (https://www.learncpp.com/cpp-tutorial/assert-and-static_assert/)[5]) can be used instead (or also) (as asserts are intended to document things that should never happen):

```cpp
#include <iostream>
#include <cassert>

void print(const int* ptr) // now a pointer to a const int
{
    assert(ptr); // fail the program in debug mode if a null pointer is passed (since this should never happen)

    // (optionally) handle this as an error case in production mode so we don't crash if it does happen
    if (!ptr)
        return;

    std::cout << *ptr << '\n';
}

int main()
{
    int x{ 5 };

    print(&x);
    print(nullptr);

    return 0;
}
```

Prefer pass by (const) reference

Note that function `print()` in the example above doesn't handle null values very well -- it effectively just aborts the function. Given this, why allow a user to pass in a null value at all? Pass by reference has the same benefits as pass by address without the risk of inadvertently dereferencing a null pointer.

Pass by const reference has a few other advantages over pass by address.

First, because an object being passed by address must have an address, only lvalues can be passed by address (as rvalues don't have addresses). Pass by const reference is more flexible, as it can accept lvalues and rvalues:

```cpp
#include <iostream>

void printByValue(int val) // The function parameter is a copy of the argument
{
    std::cout << val << '\n'; // print the value via the copy
}

void printByReference(const int& ref) // The function parameter is a reference that binds to the argument
{
    std::cout << ref << '\n'; // print the value via the reference
}

void printByAddress(const int* ptr) // The function parameter is a pointer that holds the address of the argument
{
    std::cout << *ptr << '\n'; // print the value via the dereferenced pointer
}

int main()
{
    printByValue(5);     // valid (but makes a copy)
    printByReference(5); // valid (because the parameter is a const reference)
    printByAddress(&5);  // error: can't take address of r-value

    return 0;
}
```

Second, the syntax for pass by reference is natural, as we can just pass in literals or objects. With pass by address, our code ends up littered with ampersands (&) and asterisks (*).

In modern C++, most things that can be done with pass by address are better accomplished through other methods. Follow this common maxim: "Pass by reference when you can, pass by address when you must".

Best practice

Prefer pass by reference to pass by address unless you have a specific reason to use pass by address.

Next lesson
12.11 Pass by address (part 2)

6

Back to table of contents

7

Previous lesson
12.9 Pointers and const

8

9

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/[11] are connected to your provided email address.

321 COMMENTS

Newest

Balaji Kannigesvaran

March 9, 2025 7:26 pm PDT

Prefer pointer-to-const function parameters over pointer-to-non-const function parameters, unless the function needs to modify the object passed in.
Do not make function parameters const pointers unless there is some specific reason to do so.

The above statements seems contradicting. I couldn't get this

0 Reply

Copernicus

Reply to Balaji Kannigesvaran [12] May 19, 2025 8:32 pm PDT

1. Prefer a pointer to const parameter(a variable that cant be modified)

```cpp
void doSomething(const int* ptr)
{
    //do some stuff
}
```

prefer the above, unless you need the function to modify the parameter.

2. Don't make the pointer itself const(the pointer cant be changed where it is pointing to).

```cpp
void doSomething(int* const ptr)
{
    //do some stuff
}
```

Last edited 1 month ago by Copernicus

2 Reply

Grace
February 21, 2025 2:10 pm PST

I don't like this "pass by address" notion. I think it only serves to create further confusion. There are only two options: pass by value and pass by reference. What you're calling pass by address is still pass by value, where the value being passed just happens to be a pointer. You are passing a pointer by value. It's also possible to pass a pointer by reference, if you need to modify the value of the pointer in the caller scope. So when you have pointers as function arguments, there still exist the pass by value and pass by reference options.

1 Reply

Alex Author
Reply to Grace 13 February 26, 2025 11:17 pm PST

I don't like pass by address either, but it exists and people use it, therefore we cover it. :)

Also see the last section of https://www.learncpp.com/cpp-tutorial/pass-by-address-part-2/

2 Reply

Grace
Reply to Alex 14 June 19, 2025 11:36 pm PDT

"I don't like pass by address either, but it exists" I disagree. What I'm saying is that it does not exist. It makes no sense to say pass BY address. What you're doing is passing AN address (the pointer) BY value (the method). You can also pass AN address BY reference (reference to a pointer). By value and by reference are the two options, there is no third. When pointers get involved, it's nothing more than a data type that's still being either passed by value or by reference. Maybe there would be less confusion if pass by value was called pass by copy - which is more descriptive of what's actually happening under the hood.

Last edited 12 days ago by Grace

0 Reply

thoup
Reply to Grace 13 February 22, 2025 2:54 pm PST

As far as I know, pass by address is generally not used in C++ and pass by reference is preferred. It may only exist in C++ because that is how it's done in C, and C++ is a superset of C

1 Reply

ShyVids
Reply to thoup 15 March 24, 2025 10:40 am PDT

actually, i think, is the only way to pass address of objects in the heap so that's why exist

0 Reply

NordicPeace
December 17, 2024 8:23 am PST

Quick revision from Chat-gpt :
Explanation of Pass by Address

`Analogy for Pass by Address`

```
Imagine you want to fix something in your friend's house.

Pass by Value: Your friend gives you a blueprint (copy) of their house. You make changes
to the blueprint, but their actual house remains unchanged.
Pass by Address: Your friend gives you the key to their house (address). You directly go
into their house and make changes. These changes are seen in the actual house.

In programming:
```

`The "key" is the memory address.`
`The "house" is the variable in memory.`
`When you have the "key" (pass by address), you can directly modify the house (variable).`

0 Reply

mikewhocheeseharry
September 8, 2024 2:48 pm PDT

I dont understand why if (!ptr) return ; is better than if (ptr){}. They both check ptr once and ptr value might change in the rest of the code for both of them. I think I might have to add check conditions in the first one as well.

0 Reply

Alex Author
Reply to mikewhocheeseharry 16 September 9, 2024 10:31 pm PDT

It's called the bouncer pattern. It puts all your error cases up front and if any error occur, you are bounced out of the function. This also means nothing needs to be nested.

If you use if (ptr) {} then your happy path gets nested one level. If you have multiple conditions that need to be tested, you can end up with your happy path nested quite deeply.

See https://www.learncpp.com/cpp-tutorial/assert-and-static_assert/

6 Reply

Strain
February 6, 2024 6:53 pm PST

Ok, so, here's a rule I made for myself:

Won't change it? Const reference.
Will change it? Pointer.

I guess it would make the goal of changing the value more explicit.

3 Reply

Alex Author

Reply to Strain 17 February 7, 2024 12:10 pm PST

Personally, I prefer pass by non-const reference and a good function name in cases where nullptr is not a valid argument. This avoids the possibility of nullptr dereferences.

4 Reply

Zija

February 1, 2024 7:51 am PST

The parameter `ptr` can be made to a const pointer (i.e. `const std::string* const ptr`) since its address isn't going to change, no?

```cpp
void printByAddress(const std::string* ptr) // The function parameter is a pointer that holds the address of str
{
    std::cout << *ptr << '\n'; // print the value via the dereferenced pointer
}
```

Last edited 1 year ago by Zija

0 Reply

Alex Author

Reply to Zija 18 February 2, 2024 10:00 am PST

We could, but we typically don't for the same reason we don't make value parameters const. If the function wants to change `ptr` for some reason, why stop it? It won't affect the caller.

3 Reply

Swaminathan R

Reply to Alex 19 May 18, 2024 7:32 am PDT

But it doesn’t make the intention clear correct? If we made it as const int* const ptr, then we know that we don’t want to change memory address…

0 Reply

Alex Author

Reply to Swaminathan R [20] May 18, 2024 9:09 pm PDT

The question is whether demonstrating the intention is valuable enough to offset the extra clutter in your code. This kind of const-ing makes no meaningful promise to the caller, who is the primary user of the function's interface.

It does help other readers of the code understand that the pointer will not change to point at something else. That might be helpful on a large team.

Personally, I don't think it's worth doing, but that's a judgement call and plenty of devs think otherwise.

1 Reply

Zykee

January 30, 2024 7:00 pm PST

```
void printByAddress(const std::string* ptr)
{
    std::cout << *ptr << '\n';
}
```

When I use " std::string* ptr(&str);" ,is there an implicit convertion from std::string* to const std::string* and it`s allowed?
I bet I had seen this knowledge point somewhere...

Last edited 1 year ago by Zykee

0 Reply

Alex Author

Reply to Zykee [21] February 1, 2024 12:42 pm PST

Yes. This is called a qualification conversion. See https://en.cppreference.com/w/cpp/language/implicit_conversion

3 Reply

thanks

November 8, 2023 1:42 am PST

ok so i guess references are pointers but better(/safer)!

3 Reply

NordicPeace

Reply to thanks [22] December 17, 2024 5:43 am PST

yup, go for reference

0 Reply

le batte
October 11, 2023 12:17 am PDT

```
int main()
{
    int x{ 5 };
    print(&x);

    int* myPtr {};
    print(myPtr);

    return 0;
}
```

`When this program is run, it will print the value 5 and then most likely crash.`

isn't that supposed to print the address of x?

0 Reply

Alex Author
Reply to le batte [23] October 13, 2023 11:18 pm PDT

No, we pass the address of `x` to `print()`, and then `print()` dereferences that address to get the value at that address.

1 Reply

xx yy
October 1, 2023 8:07 am PDT

Just wanted to ask, surely this is still passing by value, it's just that instead we are passing the pointer by value?

1 Reply

Alex Author
Reply to xx yy [24] October 2, 2023 1:21 pm PDT

Yes, at the end of the day, everything is implemented as pass by value. This is noted in the next lesson.

0 Reply

Swaminathan R
Reply to Alex [25] May 18, 2024 7:37 am PDT

But references?

```
int fcn(int& a) {…}
```

If the call this function like this:

```
fcn(x);
```

then what is passed by value here? My understanding was only thar “a” will now be an alias for “x”

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/introduction-to-function-parameters-and-arguments/
3. https://www.learncpp.com/cpp-tutorial/pass-by-lvalue-reference/
4. https://www.learncpp.com/cpp-tutorial/constant-variables-named-constants/
5. https://www.learncpp.com/cpp-tutorial/assert-and-static_assert/
6. https://www.learncpp.com/cpp-tutorial/pass-by-address-part-2/
7. https://www.learncpp.com/
8. https://www.learncpp.com/cpp-tutorial/pointers-and-const/
9. https://www.learncpp.com/pass-by-address/
10. https://www.learncpp.com/cpp-tutorial/inline-functions-and-variables/
11. https://gravatar.com/
12. https://www.learncpp.com/cpp-tutorial/pass-by-address/#comment-608419
13. https://www.learncpp.com/cpp-tutorial/pass-by-address/#comment-607955
14. https://www.learncpp.com/cpp-tutorial/pass-by-address/#comment-608103
15. https://www.learncpp.com/cpp-tutorial/pass-by-address/#comment-607981
16. https://www.learncpp.com/cpp-tutorial/pass-by-address/#comment-601764
17. https://www.learncpp.com/cpp-tutorial/pass-by-address/#comment-593346
18. https://www.learncpp.com/cpp-tutorial/pass-by-address/#comment-593116
19. https://www.learncpp.com/cpp-tutorial/pass-by-address/#comment-593162
20. https://www.learncpp.com/cpp-tutorial/pass-by-address/#comment-597238
21. https://www.learncpp.com/cpp-tutorial/pass-by-address/#comment-593048
22. https://www.learncpp.com/cpp-tutorial/pass-by-address/#comment-589527
23. https://www.learncpp.com/cpp-tutorial/pass-by-address/#comment-588519
24. https://www.learncpp.com/cpp-tutorial/pass-by-address/#comment-588019
25. https://www.learncpp.com/cpp-tutorial/pass-by-address/#comment-588114
26. https://g.ezoic.net/privacy/learncpp.com