

7.4 — Passing arguments by address

BY ALEX ON JULY 25TH, 2007 | LAST MODIFIED BY ALEX ON JULY 7TH, 2017

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:

```
1  #include <iostream>
2
3  void foo(int *ptr)
4  {
5      *ptr = 6;
6  }
7
8  int main()
9  {
10     int value = 5;
11
12     std::cout << "value = " << value << '\n';
13     foo(&value);
14     std::cout << "value = " << value << '\n';
15     return 0;
16 }
```

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:

```
1  void printArray(int *array, int length)
2  {
3      for (int index=0; index < length; ++index)
4          std::cout << array[index] << ' ';
5  }
```

Here is an example program that calls this function:

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

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:

```
1  void printArray(int *array, int length)
```

```

2   {
3       // if user passed in a null pointer for array, bail out early!
4       if (!array)
5           return;
6
7       for (int index=0; index < length; ++index)
8           cout << array[index] << ' ';
9   }
10
11  int main()
12  {
13      int array[6] = { 6, 5, 4, 3, 2, 1 };
14      printArray(array, 6);
15  }

```

Passing by const address

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

```

1  void printArray(const int *array, int length)
2  {
3      // if user passed in a null pointer for array, bail out early!
4      if (!array)
5          return;
6
7      for (int index=0; index < length; ++index)
8          std::cout << array[index] << ' ';
9  }
10
11  int main()
12  {
13      int array[6] = { 6, 5, 4, 3, 2, 1 };
14      printArray(array, 6);
15  }

```

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 by address, 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.

```

1  #include <iostream>
2
3  void setToNull(int *tempPtr)
4  {
5      // we're making tempPtr point at something else, not changing the value that tempPtr points to.
6      tempPtr = nullptr; // use 0 instead if not C++11
7  }
8
9  int main()
10 {
11     // First we set ptr to the address of five, which means *ptr = 5
12     int five = 5;
13     int *ptr = &five;
14
15     // This will print 5
16     std::cout << *ptr;
17
18     // tempPtr will receive a copy of ptr
19     setToNull(ptr);
20
21     // ptr is still set to the address of five!

```

```

22
23 // This will print 5
24 if (ptr)
25     std::cout << *ptr;
26 else
27     std::cout << " ptr is null";
28
29 return 0;
30 }

```

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:

```

1  #include <iostream>
2
3  void setToSix(int *tempPtr)
4  {
5      *tempPtr = 6; // we're changing the value that tempPtr (and ptr) points to
6  }
7
8  int main()
9  {
10     // First we set ptr to the address of five, which means *ptr = 5
11     int five = 5;
12     int *ptr = &five;
13
14     // This will print 5
15     std::cout << *ptr;
16
17     // tempPtr will receive a copy of ptr
18     setToSix(ptr);
19
20     // tempPtr changed the value being pointed to to 6, so ptr is now pointing to the value 6
21
22     // This will print 6
23     if (ptr)
24         std::cout << *ptr;
25     else
26         std::cout << " ptr is null";
27
28     return 0;
29 }

```

This prints:

56

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:

```
1  #include <iostream>
2
3  // tempPtr is now a reference to a pointer, so any changes made to tempPtr will change the argument as
4  // well!
5  void setToNull(int *&tempPtr)
6  {
7      tempPtr = nullptr; // use 0 instead if not C++11
8  }
9
10 int main()
11 {
12     // First we set ptr to the address of five, which means *ptr = 5
13     int five = 5;
14     int *ptr = &five;
15
16     // This will print 5
17     std::cout << *ptr;
18
19     // tempPtr is set as a reference to ptr
20     setToNull(ptr);
21
22     // ptr has now been changed to nullptr!
23
24     if (ptr)
25         std::cout << *ptr;
26     else
27         std::cout << " ptr is null";
28
29     return 0;
30 }
```

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 on [references](#), 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) comes *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!

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 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 and dereference the pointer argument).
- 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.

Rule: Prefer pass by reference to pass by address whenever applicable.



7.4a -- Returning values by value, reference, and address



Index



7.3 -- Passing arguments by reference

Share this:



[C++ TUTORIAL](#) | [PRINT THIS POST](#)

133 comments to 7.4 — Passing arguments by address

[« Older Comments](#) [1](#) [2](#)



Sagar
[June 29, 2018 at 4:34 am · Reply](#)

When passing by address, is argument copy made as a function parameter?

nascardriver
[June 29, 2018 at 5:03 am · Reply](#)