7.4 — Passing arguments by address

BY ALEX ON JULY 25TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

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:

```
#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:

```
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:

```
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
}
```

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:

```
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)
        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:

```
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 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.

```
#include <iostream>

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

```

```cpp
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:

```
#include <iostream>

// tempPtr is now a reference to a pointer, so any changes made to tempPtr will change the arg
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 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!

Pass by address makes modifiable parameters explicit

Consider the following example:

```
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 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

C++ TUTORIAL | PRINT THIS POST

188 comments to 7.4 — Passing arguments by address

« Older Comments 1 2 3

chai
January 16, 2020 at 12:00 pm · Reply

Sorry! It should read.

Also, the above narrative states that passing by address and dereferencing a pointer is generally slower than the process of passing by value. How does this compare to the process of passing by reference.

chai
January 16, 2020 at 11:58 am · Reply

So for clarity, if a function needs to return (/change) 2 values (I cannot think of a practical example yet, but just in case) which is the best method?
Pass by address with dereferencing nullptr risk. Or by reference , using "out" suffix, but at a risk of looking like passing by value.

Also, the above narrative states that passing by value and dereferencing a pointer is generally slower than the process of passing by value. How does this compare to the process of passing by reference.

Thanks.

nascardriver
January 17, 2020 at 1:22 am · Reply

> which is the best method?
I can't answer that question. If you're passing by pointer, you should make sure that a `nullptr` doesn't cause issues.

Pointers and references are the same under the hood.

hellmet
October 23, 2019 at 3:43 am · Reply

```
1  // Instead of having this and the call like so,
2  void setToNull(int *&tempPtr)
3  {
4      tempPtr = nullptr; // use 0 instead if not C++11
5  }
6  ...
7  setToNull(someptr);
8
9  // It is equivalent to the following right?
10 void setToNull(int **tempPtr)
11 {
12     tempPtr = nullptr; // use 0 instead if not C++11
13 }
14 ...
15 setToNull(&someptr);
16 ...
```

Since as mentioned in the lesson, references are just implemented with pointers under the hood, are references just syntactic sugar so that I don't have to call the function with the ampersand (as in line 15)?

nascardriver
October 23, 2019 at 4:11 am · Reply

If you change line 12 to

```
*tempPtr = nullptr;
```

then both are the same. There are small differences, like references not being able to be null. Mostly it's a difference in syntax.

hellmet
October 23, 2019 at 4:19 am · Reply

Ahh yes line 12, my bad!
So references vs address is more or less a preference and ease of use right (apart from the slight differences as you reminded me, ref can't be null)?

nascardriver
October 23, 2019 at 4:24 am · Reply

Yes.
References are primarily used when something is passed by const reference to prevent a copy.
I don't usually pass by non-const reference, as it can be misleading. Take line 7 for example.
The caller can't know that `someptr` can be modified without looking at `setToNull`. Line 15 on the other hand is pretty obvious.

hellmet
October 23, 2019 at 5:24 am · Reply

Okay perfect!

A small clarification ...
The functions are effectively documented in their respective header files. So a diligent programmer should look at the function declaration and if it does not have a const parameter, they should expect that the function might change the arguments, right? Say I was to make a small library, the reference syntax is much cleaner. Would preferring the reference syntax make it less 'C++' style or unintuitive?

nascardriver
October 23, 2019 at 5:34 am · Reply

> header
Yes, it's clear from looking at the header. But you'd have to look at the header. Passing by pointer is clearer even to someone who doesn't know the function.

> Would preferring the reference syntax make it less 'C++' style
There's no such thing. Some other languages have a uniform style. In C++ the style varies between programmers and projects.

> Would preferring the reference syntax make it [...] unintuitive?
I'd say so, yes. When I see a function call like line 7, my intuition tells me that the argument doesn't get modified.
In practice, modifying arguments is rare. You'll learn about classes later, they allow an object to be modified by using a member function, eg.

```
// Here @tempPtr isn't a pointer. It's obvious what's happening and
// we don't have to worry about pointer vs reference.
```

```
tempPtr.setToNull();
```

hellmet
October 23, 2019 at 7:18 am · Reply

Thank you!

Pavel Y
September 4, 2019 at 5:53 pm · Reply

I understand that I can use a for loop instead, however, is this a viable alternative, or is it prone to error. Lets assume all of the values in our array are not zero.

```
1   void print_pointer(int *&input)  // prints the values of an array.
2   {
3     int i{0};
4
5     while (input[i] != 0)
6     {
7       std::cout << &input[i] << ' ' << input[i] << std::endl;
8       i = i + 1;
9     }
10    cout << endl;
11  }
```

nascardriver
September 5, 2019 at 12:35 am · Reply

\`input\` should be a \`const int* input\` or \`const int input[]\`. Don't pass a pointer by reference unless you modify it.
Don't use \`using namespace std;\`.
Line 8 should be \`++i\`.
Don't use \`std::endl\` unless you need to flush the stream. Most of the time '\n' suffices.

Use a for-loop if you know how often your loop is going to run. You don't know that, so a while-loop is correct. You don't need an index though.

```
void print_array(const int* arr)
{
  while (*arr != 0)
  {
    std::cout << arr << ' ' << *arr << '\n';
    ++arr;
  }
}
```

Pavel Y
September 4, 2019 at 5:16 pm · Reply

Am I understanding this correctly?

```
void print(int* input)  //  input should be a pointer.

void print(int *input)  //  Same as previous.

void print(int &input)  //  input should be a reference to a value.

void print(int* &input)  //  input should be a reference to a pointer.

void print(int *&input)  //  same as previous.
```

nascardriver
September 5, 2019 at 12:28 am · Reply

yes

Nirbhay
August 7, 2019 at 11:48 pm · Reply

Why doesn't this code print "NULL FOUND" ?

```
#include <iostream>

void printArray(int *array, int length)
{
    // if user passed in a null pointer for array, bail out early!
    if (!array)
        std::cout << "NULL FOUND";
        return;
    for (int index=0; index < length; ++index)
        std::cout << array[index] << ' ';
}

int main()
{
    int array[] = {};
    printArray(array, 1);
    return 0;
}
```

Alex
August 8, 2019 at 8:03 pm · Reply

The array in main decays to a pointer when passed to printArray, pointing to the first element of the array, which has a non-zero address. So the array parameter of printArray is non-zero.

« Older Comments 1 2 3