7.4 — Passing arguments by address

BY ALEX ON JULY 25TH, 2007 | LAST MODIFIED BY ALEX ON FEBRUARY 15TH, 2016

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

The above snippet prints:

```
value = 5
value = 6
```

As you can see, the function foo() changed the value of the argument value through the pointer parameter ptr.

Pass by address is typically used with arrays and dynamically allocated variables. 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          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 };
4      printArray(array, 6);
5  }
```

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)
        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 passed by value

When you pass an 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
s to.
    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;
```

```
18
19      // tempPtr will receive a copy of ptr
20      setToNull(ptr);
21
22      // ptr is still set to the address of five!
23
24      // This will print 5
25      if (ptr)
26          std::cout << *ptr;
27      else
28          std::cout << " ptr is null";
29
30      return 0;
    }
```

tempPtr receives a copy of the address of ptr. 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 receives a copy of the address from the argument. At this point, the function parameter and the argument have the same address, so they both point to the same value.
- If the function parameter is *assigned* to a different address, that doesn't impact the argument, since the function parameter is a copy.
- If the function parameter is *dereferenced* to change the value being pointed to, that will impact the value the argument is pointing to, since it's the same value!

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
```

Passing addresses by reference

The next logical question is, “What if we want to be able to change the address of an argument 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
ument as well!
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 address 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 **passing arguments by reference**, 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 value 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!

Summary

Advantages of passing by address:

- It 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.
- 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.

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 pointer values.
- When passing built-in arrays (if you're okay with the fact that they'll decay into a pointer).

When not to use pass by address:

- 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

45 comments to 7.4 — Passing arguments by address

Jason
December 19, 2007 at 5:45 pm · Reply

Disadvantages of passing by address:

Because dereferencing a pointer is slower than accessing a value directly, accessing arguments passed by address is slower than accessing arguments passed by value.

Could that sentence possibly supposed to read as follows:

Because dereferencing a pointer is slower than accessing a value directly, accessing arguments passed by address is slower than accessing arguments passed by reference.

I would think (am assuming) that arguments passed by value would take the longest because they have to be copied.

[It turns out that references are usually implemented by the compiler (under the hood) using pointers. Consequently, references aren't any faster than addresses. They just have a nicer syntax, and are safer to use. -Alex]

Jason
December 19, 2007 at 11:31 pm · Reply

I think I get it. Passing by value is slow because it does a copy. Passing by reference or address is basically the same thing, just different syntax.

Comparing these two programs:

```
1  void foo(int *pValue)
2  {
3      *pValue = 6;
4  }
5
6  int main()
7  {
8      int nValue = 5;
9
10     cout << "nValue = " << nValue << endl;
11     foo(&nValue);
12     cout << "nValue = " << nValue << endl;
```

```
    return 0;
}

Ouput:
nValue = 5
nValue = 6
```

```
void func(char *address)
{
    address = "b";
}

int main()
{
    char *array = "a";
    cout << "array = " << array << endl;
    func(array);
    cout << "array = " << array << endl;
}

Output:
array = a
array = a
```

Why doesn't the second example act as the first example. As the output would be:

```
array = a
array = b
```

What I've noticed is when passing an array by address it changes the address that the pointer points to when inside the function. This causes the original value to never be changed. I guess that's just how it works.

Any insight on this issue? Btw, I know that char *array is not an array but a pointer, but once again it really is an array when used with strings right? Just different syntax right?

Alex
October 25, 2015 at 2:12 pm · Reply

> I think I get it. Passing by value is slow because it does a copy. Passing by reference or address is basically the same thing, just different syntax.

Passing by value can be slow because it makes a copy. If you're passing a fundamental type (e.g. an int or a double), there's no problem. But if you're passing a large struct or class, making a copy can be very expensive. Passing by address (or reference) makes a copy of the address of the argument, instead of copying the entire thing, which can be much faster.

> What I've noticed is when passing an array by address it changes the address that the pointer points to when inside the function. This causes the original value to never be changed. I guess that's just how it works.

Good question and insight. As it turns out, pointer parameters are actually passed by value. Consequently, if you try to change what a pointer points to inside a function, it's the same as changing a variable locally -- as soon as you leave the function, it will revert back to what it was. However, if you dereference the pointer and change the value of what it points to, that won't be reverted. If you actually want to be able to change the address that a pointer points to inside a function, the best way to do this is to pass the pointer itself by reference:

void func(char *&address)

Incidentally, this line of code is dangerous:

address = "b"

This is setting the address of the "address" variable to the address of "b". What is the address of "b"? "b" isn't a variable, so this doesn't really make any sense. It probably works in this case due to the way the compiler is dealing with string literals, but I certainly wouldn't want to trust it.

> Btw, I know that char *array is not an array but a pointer, but once again it really is an array when used with strings right? Just different syntax right?

*array in this case is just a pointer. Since it points to an array, you can use it like an array, but it isn't an array itself.

Jason
December 20, 2007 at 1:47 pm · Reply

Awesome, I get it.

I'd just like to say, that all made me realize that communicating C++'s logic is an artform.

Tom
March 19, 2008 at 10:15 pm · Reply

1. In your first example function, PrintArray, don't you have to dereference the pnArray variable, like this??:

```
1 void PrintArray(int *pnArray, int nLength)
2 {
3     for (int iii=0; iii < nLength; iii++)
4         cout << *pnArray[iii] << endl;
5 }
```

2. Minor typo: "The next logical question is, â€œWhat if we want **to be able to be able to** change the address"

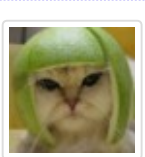
Alex
March 20, 2008 at 7:26 am · Reply

You don't need to do an explicit dereference when using the array index operator ([]) because the array index operator does an implicit dereference.

weirdolino
October 16, 2008 at 10:08 pm · Reply

I'm having trouble understanding the conclusions. Under advantages you say
Because a copy of the argument is not made, it is fast, even when used with large structs or classes.
and under disadvantages you say
Because dereferencing a pointer is slower than accessing a value directly, accessing arguments passed by address is slower than accessing arguments passed by value.

Is passing by reference/address slower or faster than by value? Does it depend on the size of the data passed to the function?



Alex
October 17, 2008 at 11:55 am · Reply

When we talk about speed and efficiency with regards to variables, we're talking about two separate things:

1) The speed of creating the variable and initializing it's values. This cost is paid once when the variable is created, and can be significant for large classes that are copied by value. This is why large classes are better passed by pointer or reference than by value, especially if a function is called many times.

2) The speed of accessing the value stored in the variable. This cost is paid each time the variable is accessed. Although accessing pointers and references is slower than accessing normal variables because of the

dereference that takes place, in practice this is rarely a large concern unless you have some kind of loop that accesses the variable hundreds or thousands of times.

weirdolino
October 17, 2008 at 9:35 pm · Reply

I think I understand the difference now. This seems to be a rather advanced topic. Thanks for the quick reply!

Kukudum
January 30, 2009 at 1:07 pm · Reply

In the example you used:
The function |
V

```
1  void PrintArray(int *pnArray, int nLength)
2  {
3      for (int iii=0; iii < nLength; iii++)
4          cout << pnArray[iii] << endl;
5  }
6
7
8  //Here is an example program that calls this function:
9
10 int main()
11 {
12     int anArray[6] = { 6, 5, 4, 3, 2, 1 };
13     PrintArray(anArray, 6);
14 }
```

Isn't PrintArray(anArray, 6); supposed to be PrintArray(&anArray, 6); or am I missing something o.O

Alex
February 6, 2009 at 12:19 am · Reply

You are missing the fact that built-in arrays in C++ will decay into a pointer to the array when passed to a function. Since PrintArray() is expecting a pointer for its first parameter, once anArray decays into a pointer, we're all set.

Gareth37
April 7, 2009 at 2:19 pm · Reply

in ch 6.6 you have this -
This would be the conceptual equivalent of the following nonsensical example:

```
1  int anArray[] = { 3, 5, 7, 9 };
2  anArray = 8; // what does this mean?
```

so how comes in the below example we can do

```
1  PrintArray(anArray, 6); ?
```

why isnt it PrintArray(anArray[], 6);
or PrintArray(anArray[0], 6); the pointer to its first element ?
i know these dont work because it wont compile
but it just seems odd we are referring to arrays as

anArray[] and then it becomes just anArray ?

```
void PrintArray(int *pnArray, int nLength)
{
    for (int iii=0; iii < nLength; iii++)
        cout << pnArray[iii] << endl;
}

//Here is an example program that calls this function:

int main()
{
    int anArray[6] = { 6, 5, 4, 3, 2, 1 };
    PrintArray(anArray, 6);
}<!--formatted-->
```

Alex
May 1, 2009 at 8:55 pm · Reply

When we declare a pointer, we use the * or [] syntax to let the compiler know that this variable is a pointer as opposed to a normal variable. However, from that point forward, we can just use the variable's name and the compiler already knows it's a pointer.

So if we want to pass the array to a function that expects a pointer, all we need to do is pass the variable itself.

It's the same as the following:

```
int printPtr(int *pInt)
{
   // pInt is a pointer, so we dereference it to get it's value
   std::cout << *pInt;
}

int nX = 5;
int *pX = &nX; // pX is a pointer that points to nX

printPtr(pX); // we pass the pointer here
```

pX already knows it's a pointer, so when we pass it to printPtr, we don't need to tell the compiler that it's a pointer again.

jeremy
April 21, 2009 at 12:43 am · Reply

hi alex!
i just want to clarify some points…im quite confused with DEREFERENCING and EVALUATING pointers. correct me if im wrong:

1. my idea is that in below statement, we are assigning the pointer to POINT to the address of nFive
int *pPtr = &nFive;
2. on this statement:
cout << *pPtr;
we are DEREFERENCING the pointer?
Kindly clarify.
Thanks a lot

Alex
May 1, 2009 at 9:40 pm · Reply

You are correct. We dereference a pointer to get the value of what it's pointing at.

int *pPtr = &nFive; // pPtr holds the address of nFive
cout << pPtr; // prints the address of nFive cout << *pPtr; // prints the value at the address of nFive, which is whatever value was assigned to nFive

Bob
November 22, 2009 at 6:31 pm · Reply

Excuse me but I don't understand the difference between

```
1 void SetToSix(int *&pTempPtr)
2 {
3 pTempPtr = &nSix
4 }<!--formatted-->
```

and

```
1 void SetToSix(int *pTempPtr)
2 {
3 *pTempPtr = nSix
4 }
```

Don't they both do the same thing? Why bother with the top one (which is more complicated in my opinion) when you can just do the second one?

Daniel
November 24, 2009 at 12:25 am · Reply

Let me give an example:

```
1  #include <iostream>
2
3  void PassPointerByValue1(int * thing) {
4      *thing = 1;
5  }
6
7  void PassPointerByValue2(int * thing) {
8      static int x = 2;
9      thing = &x;
10 }
11
12 void PassPointerByReference3(int * &thing) {
13     *thing = 3;
14 }
15
16 void PassPointerByReference4(int * &thing) {
17     static int x = 4;
18     thing = &x;
19 }
20
21 int main(int argc, char * * argv) {
22     int y = 0;
23     int * stuff = &y;
24     std::cout << *stuff << std::endl;
25
26     PassPointerByValue1(stuff);
27     std::cout << *stuff << std::endl;
28
29     PassPointerByValue2(stuff);
30     std::cout << *stuff << std::endl;
31
32     PassPointerByReference3(stuff);
33     std::cout << *stuff << std::endl;
34
```

```
    PassPointerByReference4(stuff);
    std::cout << *stuff << std::endl;
    return 0;
}<!--formatted-->
```

The output of this program is “01134” not “01234” as you might expect. This is because in “PassPointerByValue2” only a local copy of the pointer “thing” is changed whereas in “PassPointerByReference4” the actual pointer “stuff” is changed. Because a pass by reference was used, “thing” was “stuff” not a copy of “stuff” it was “stuff”. I hope that helps you see the difference. (^.^)

ming
January 28, 2015 at 2:54 pm · Reply

This example is very good for study passing pointer in different cases. I spent a hour to get understand it and finally feel comfortable about pointer.
Here is my explanation of this code:

```
#include "stdafx.h"
#include
using namespace std;

void PassPointerByValue1(int * thing) {   // int *thing is same address as stuff
address
    *thing = 1;                           // now the value in thing(same as stuf
f) address become 1
}


void PassPointerByValue2(int * thing) {  // int *thing is same address as stuff a
ddress
    static int x = 2;
    thing = &x;                          // thing address become x address ------
----stuff and thing become different address

    cout << *thing << endl;              // value in thing address is same value
in x address, BUT not stuff
}


void PassPointerByReference3(int * &thing) {   // int *thing is not just same add
ress as stuff address, int *thing IS the stuff address
    *thing = 3;                                // value in thing(which is stuff)
address become 3
}


void PassPointerByReference4(int * &thing) {   // int *thing is not just same add
ress as stuff address, int *thing IS the stuff address
    static int x = 4;
    thing = &x;                                // thing address which mean stuff
address become x's address
}


int _tmain(int argc, _TCHAR* argv[])
{


    int y = 0;
    int * stuff = &y;            // stuff address = y's address   *stuff is whate
ver value in stuff address
    cout << *stuff << endl;      //  call out 0   because this is the value in st
uff address

```

```
    PassPointerByValue1(stuff);  // copy stuff address to the function
    cout << *stuff << endl;      // call out 1    because value in stuff address
become 1


    PassPointerByValue2(stuff);  // copy stuff address to the function
    cout << *stuff << endl;      // call out 1    because value in stuff address
is still 1


    PassPointerByReference3(stuff); // copy stuff address to the function
    cout << *stuff << endl;         // call out 3   because value in stuff addres
s is 3


    PassPointerByReference4(stuff); // copy stuff address to the function
    cout << *stuff <> zz;
    return 0;
}
```

Hope this is useful 😀

Cuacharpanas
January 29, 2010 at 5:01 pm · Reply

If string literals don't have addresses what happen when I write this code which runs without errors from the compiler:

```
char *str="This is a literal string!";
cout << str;
```

What str points to?? Moreover, it works the same when a literal string is passed as an argument to a function:

display("Important message!");

where display definition is:

```
void display(char *str)
{
   cout << "The string is: " << str;
}
```

someplace str must point to, I think. The prototype void display(char str[]) also works, what is the difference??, may be str is considered as an array, however i tried to change the location it points to and get no error, It worked!

Hope for replies, Thanks a lot and congratulations for the tutorial, it's clear and extensive.

Alex
October 25, 2015 at 2:25 pm · Reply

string literals do have addresses, so your examples will work fine. However, string literals may reside in read-only memory, so don't try and change the contents of a string literal, or your program may terminate with an access violation.

Intelus
February 12, 2010 at 3:24 pm · Reply

I'd like to ask a question. In this post (7.4), you say:

Note that the length of the array must be passed in as a parameter, because arrays don't keep track of how long they are. Otherwise the PrintArray() function would not know how many elements to print.

Examine this short program (explanation below):

```
#include <iostream>

using namespace std;

void function (int*);

int main()
{

    int anArray[6] = { 6, 5, 4, 3, 2, 1 };
    cout << sizeof(anArray) << endl;
    function(anArray);

    return 0;
}

void function(int *pnArray)
{
     cout << sizeof(pnArray) << endl;
}
```

This prints:

```
24
4
```

Now, apparently, main() is somehow aware of the fact that anArray is an array with 6 elements while function() sees nothing more than a single address (pointer). Changing the data type yields the same result - pnArray is always 4 bytes (as pointers should be on 32-bit computers) while sizeof(anArray) returns the size of the array's elements put together (ex. 12 bytes for short, assuming there are 6 elements).

So I'm wondering, why does main() consider it an array and where is this information stored?

By the way, regarding the PrintArray function… Wouldn't it be better to write something like " if (anArray) PrintArray(…);" inside main() instead of making a null pointer check inside PrintArray? Time is wasted on making a function call which won't do anything. If it makes the code too complex, you can always use an inline function.

bla
August 13, 2010 at 8:27 am · Reply

I would like to know the answer to your (first) question too. Because s. th. like:

```
int nLength= sizeof(pnArray)/sizeof(int);
```

would be nice to make the function PrintArray more elegant as it would only require one parameter:

```
void PrintArray(int *pnArray); // why not??
```

Alex
October 25, 2015 at 2:33 pm · Reply

Yes, main is aware of the fact that anArray is an array of length 6. When anArray is passed to a function, the array decays into a pointer, and the length information is lost.

I'm honestly not sure why C++ does this, and it's pretty annoying. But there are ways to work around this (e.g. pass the length separately, or use std::array).

> By the way, regarding the PrintArray function… Wouldn't it be better to write something like " if (anArray) PrintArray(…);" inside main() instead of making a null pointer check inside PrintArray? Time is wasted on making a function call which won't do anything. If it makes the code too complex, you can always use an inline function.

Yes, if you know that anArray is null, there's no point in calling PrintArray() in the first place. However, it's a function should never assume the caller is doing something reasonable, which is why it's a good idea to check for null values and handle them gracefully. Otherwise, the program could crash if the caller inadvertently does pass in a null value.

moosefetcher
January 19, 2011 at 2:20 am · Reply

Hello there.
I'm new to C++ (as will soon become obvious!), and I don't understand why the following function (the second block of code on the tutorial above) prints the CONTENTS of pnArray…

```
void PrintArray(int *pnArray, int nLength)
{
for (int iii=0; iii < nLength; iii++)
cout << pnArray[iii] << endl;
}
```

Shouldn't the 'pnArray[iii]' part be dereferenced (ie; '*pnArray[iii]')to get the values IN the pointer array? Or does it not work like that with pointer arrays? Clearly there's something I'm not getting!
Any help, well appreciated.
Thanks for all the great tutorials. You have a great way of explaining concepts step by step!

hungma
February 13, 2011 at 9:30 am · Reply

i think it is the overloaded operator.you can research it.Fun!

Matt
July 31, 2011 at 2:48 pm · Reply

"pnArray[ii]" doesn't need to be dereferenced because when using the square brackets C++ implicitly does any pointer addition needed and dereferences the variable which Alex showed in an earlier tutorial.

kriain1982
January 3, 2014 at 10:40 pm · Reply

Alex,
First of all thanks for the tutorials. But I found some thing interesting as below.

```
#include

using namespace std;
int five = 5;
int six = 6;

void check(int *temp)
{
printf("\n*temp : %d\n", *temp);
*temp = six;
printf("\n*temp : %d\n", *temp);
}

int main()
```

```
{
int *ptr = &five;
printf("\nValue in ptr : %d\n", *ptr);
check(ptr);
printf("\nValue in ptr : %d\n", *ptr);
return 0;
}
```

see the Output :

Value in ptr : 5

*temp : 5

*temp : 6

Value in ptr : 6

So, even I am sending pointer (by value), *ptr is getting modified. How can you explain this?

Alex
October 25, 2015 at 2:35 pm · Reply

Yes, this is expected. The address is passed by value, but that address can still be dereferenced to change the value being pointed to. Since both the function parameter and the argument point to this same value, if the function changes the value this way, the argument will be changed as well.

zidane3x3
January 30, 2014 at 2:49 pm · Reply

Thank so much, i used pointer like this (*&pTempPtr) for my program without understanding clearly why but now i get it 🙂 .

youssef dirani
January 25, 2015 at 7:49 am · Reply

Hello
"The syntax for doing a reference to a pointer is a little strange (and easy to get backwards"
What's the meaning of "easy to get backwards"?

Alex
January 25, 2015 at 12:07 pm · Reply

The correct syntax for a pointer reference is *&pPtr. For a non-pointer variable, you'd create a reference to it like this: &nFoo. Since a pointer is declared as *pPtr, you'd think the correct syntax for a pointer reference should be &*pPtr, with the ampersand on the outside. But it isn't so.

Abarajithan
June 17, 2015 at 7:13 am · Reply

Well, I understood something like this as a reason for why the use of ref over variables is slower for smaller datatypes. Tell me if I'm right.

In a 64 bit machine, an address is 8 bytes, while an int is 4 bytes. So, it takes more time to copy an address than to copy an int. Plus, there's the time taken for deref-ing as u mentioned. As a result, performance wise, passing by variables is better for smaller size variables while passing by ref is better for bigger structs and classes.

Another question. If array is really a pointer and if it cannot keep track of its own length, how does sizeof() return the total correct size of arrays?

Alex
October 25, 2015 at 2:55 pm · Reply

No, the difference between copying 4 and 8 bytes should be pretty trivial. The issue comes when you have a struct or class that contains a lot of data, like 1000 bytes or more. That can take a lot of time to copy.

An array isn't a pointer, but they're related. Fixed arrays know their length. However, a fixed array that has decayed into a pointer, or a pointer to a dynamic array doesn't know the array's length, because it's just a pointer.

Shivam Tripathi
July 30, 2015 at 11:59 am · Reply

```
#include<iostream>
using namespace std;
int main()
{
    volatile int* ptr;
    volatile int* ptr1;
    {
        int x=5;
        ptr=&x;
    }
    {
        int y=8;
        ptr1=&y;
    }

    cout<<ptr<<"t"<<*ptr<<"t"<<ptr1<<"t"<<*ptr1;
    return 0;
}
```

Alex…what's the analysis of this code…plz tell in-depth…O/P is 1 5 1 8….moreover when x & y are coming out of the scope how the pointers are dereferencing them…isn't they produce DANGLING POINTERS??…plz help

Alex
July 30, 2015 at 3:29 pm · Reply

The pointers will still point to the memory addresses occupied by the (now out of scope) local variables. The contents of those memory locations may still contain the values put there by the local variables -- or they may not.

So yes, this is a case where the pointers are left dangling (pointing to memory that has been deallocated).

Shivam Tripathi
July 31, 2015 at 4:31 am · Reply

hmm…got it…it means that although any variable goes out of scope but it's value may-be still accessible until the OS replaces that stack portion with some other sibling block…correct me if m wrong???

Moreover…i hv a confusion regarding the pass of the address by reference..consider this snippet:

```
 int main()
 {
   int x;                // 'x' defined.
   int* ptr=&x;          // pointer-variable 'ptr' declared..initialized with the
address of 'x'.


   foo(ptr);             // foo() called...sends the 'ptr' (address of 'x') by ref
erence.
 }

 void foo(int* &ptr2)  // 'ptr2' is a reference to the pointer-variable 'ptr'...t
hus any change
 {                     //  to it will make 'ptr' to change also.

   int y;              // new variable 'y' defined.
   ptr2=&y;            // 'ptr2' initialized with the address of 'y', this will m
ake 'ptr' to //point at 'y'

 }

/* Now after the function ends, both 'ptr2' and 'y' goes out of the scope...but s
ince the address was passed by reference thus 'ptr' (the pointer variable in call
er i.e. main()) is left out to point at 'y' which is actually gone out of scop
e...so isn't 'ptr' a DANGLING POINTER????. Will it point to a VALID location..and
if-so then what will be that address */
```

Plz..answer both sections…

Alex
July 31, 2015 at 8:47 am · Reply

Yes, ptr is now a dangling pointer, because foo changed it's address to point to local variable y, which went out of scope.

Shivam Tripathi
July 31, 2015 at 10:42 am · Reply

so it's our duty to assign "ptr" to "null"…correct???

Alex
August 1, 2015 at 10:02 am

Yes sir.

sindhu
September 1, 2015 at 4:29 am · Reply

Yes it is really good

Rohde Fischer
September 7, 2015 at 11:45 pm · Reply

Beware there be dragons! A word of warning that should probably have been in the guidelines is:

Avoid side effects whenever possible! Yes they are necessary evil from time to time, but avoiding side effects will make your code more predictable and easier to test.

Most important thing might be in a large project, a lot of function calls will be present, how can you be sure that every change that happens will be the right one? That's always a hard one, but if most of your functions do not have side effect it's at least easier (that is usually still not easy at all). Keep your code testable and predictable, it will save you headaches in the long run.

And whenever you do something with side effects, make sure to document it, even better make it apparent from the function name. Also do consider returning a new object with the result in stead, although that strategy may be troublesome on embedded devices with a limited capacity, it's easily doable in the era with 16 gigs+ ram in our computers.

Kamyar
October 17, 2015 at 7:10 am · Reply

Hi Alex
How can i pass an array to a function by reference?

Alex
October 19, 2015 at 4:42 pm · Reply

To pass an array by reference, you essentially pass it by address and then stick a reference on the front (so you're passing the address by reference).

For a dynamic array:

```
1  void myfunc(int* &array)
2  {
3
4  }
5
6  int main()
7  {
8      int *array = new int[5] { 9, 7, 5, 3, 1 };
9      myfunc(array);
10     return 0;
11 }
```

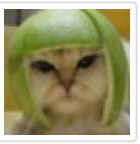
For a fixed array:

```
1  void myfunc(int (&val)[5])
2  {
3
4  }
5
6  int main()
7  {
8      int array[5] = { 9, 7, 5, 3, 1 };
9      myfunc(array);
10     return 0;
11 }
```

Sandro
February 7, 2016 at 3:07 pm · Reply

"It is always a good idea to ensure parameters passed by address are not null pointers before dereferencing them."

How can one pass a null pointer via argument to a function?



Alex
February 8, 2016 at 6:22 pm · Reply

```
1  void someFunction(int *ptr)
2  {
3  }
4
5  int main()
6  {
7      someFunction(nullptr); // use 0 instead of nullptr if not C++11
8  }
```