6.7 — Introduction to pointers

BY ALEX ON JULY 10TH, 2007 | LAST MODIFIED BY ALEX ON MARCH 14TH, 2016

In lesson **1.3 -- a first look at variables**, we noted that a variable is a name for a piece of memory that holds a value. When our program instantiates a variable, a free memory address is automatically assigned to the variable, and any value we assign to the variable is stored in this memory address.

For example:

```
int x;
```

When this statement is executed by the CPU, a piece of memory from RAM will be set aside. For the sake of example, let's say that the variable x is assigned memory location 140. Whenever the program sees the variable x in an expression or statement, it knows that it should look in memory location 140 to get the value.

The nice thing about variables is that we don't need to worry about what specific memory address is assigned. We just refer to the variable by its given identifier, and the compiler translates this name into the appropriately assigned memory address.

However, this approach has some limitations, which we'll discuss in this and future lessons.

The address-of operator (&)

The address-of operator (&) allows us to see what memory address is assigned to a variable. This is pretty straightforward:

```
#include <iostream>

int main()
{
    int x = 5;
    std::cout << x << '\n'; // print the value of variable x
    std::cout << &x << '\n'; // print the memory address of variable x

    return 0;
}
```

On the author's machine, the above program printed:

```
5
0027FEA0
```

Note: Although the address-of operator looks just like the bitwise-and operator, you can distinguish them because the address-of operator is unary, whereas the bitwise-and operator is binary.

The dereference operator (*)

Getting the address of a variable isn't very useful by itself.

The dereference operator (*) allows us to get the value at a particular address:

```
#include <iostream>

int main()
{
    int x = 5;
    std::cout << x << '\n'; // print the value of variable x
    std::cout << &x << '\n'; // print the memory address of variable x
    std::cout << *&x << '\n'; /// print the value at the memory address of variable x

    return 0;
}
```

On the author’s machine, the above program printed:

```
5
0027FEA0
5
```

Note: Although the dereference operator looks just like the multiplication operator, you can distinguish them because the dereference operator is unary, whereas the multiplication operator is binary.

Pointers

With the address-of operator and dereference operators now added to our toolkits, we can now talk about pointers. A **pointer** is a variable that holds a *memory address* as its value.

Pointers are typically seen as one of the most confusing parts of the C++ language, but they’re surprisingly simple when explained properly.

Declaring a pointer

Pointer variables are declared just like normal variable, only with an asterisk between the data type and the variable name:

```
int *iPtr; // a pointer to an integer value
double *dPtr; // a pointer to a double value

int* iPtr2; // also valid syntax (acceptable, but not favored)
int * iPtr3; // also valid syntax (but don't do this)

int *iPtr4, *iPtr5; // declare two pointers to integer variables
```

Note that the asterisk here is not a dereference. It is part of the pointer declaration syntax.

Syntactically, C++ will accept the asterisk next to the data type, next to the variable name, or even in the middle.

However, when declaring multiple pointer variables, the asterisk has to be included with each variable. It’s easy to forget to do this if you get used to attaching the asterisk to the type instead of the variable name!

```
int* iPtr6, iPtr7; // iPtr6 is a pointer to an int, but iPtr7 is just a plain int!
```

For this reason, when declaring a variable, we recommend putting the asterisk next to the variable name.

Best practice: When declaring a pointer variable, put the asterisk next to the variable name.

However, when returning a pointer from a function, it’s clearer to put the asterisk next to the return type:

```
int* doSomething();
```

This makes it clear that the function is returning a value of type int* and not an int.

Best practice: When declaring a function, put the asterisk of a pointer return value next to the type.

Just like normal variables, pointers are not initialized when declared. If not initialized with a value, they will contain garbage.

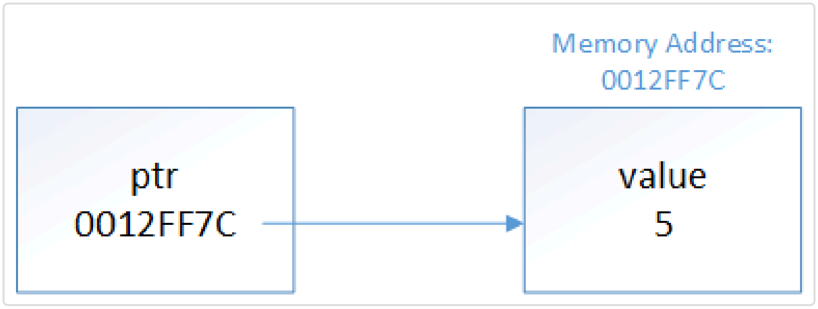
Assigning a value to a pointer

Since pointers only hold addresses, when we assign a value to a pointer, that value has to be an address. One of the most common things to do with pointers is have them hold the address of a different variable.

To get the address of a variable, we use the address-of operator:

```
int value = 5;
int *ptr = &value; // initialize ptr with address of variable value
```

Conceptually, you can think of the above snippet like this:



This is where pointers get their name from -- ptr is holding the address of variable value, so we say that ptr is “pointing to” value.

It is also easy to see using code:

```
#include <iostream>

int main()
{
    int value = 5;
    int *ptr = &value; // initialize ptr with address of variable value

    std::cout << &value << endl; // print the address of variable value
    std::cout << ptr << endl; // print the address that ptr is holding

    return 0;
}
```

On the author’s machine, this printed:

```
0012FF7C
0012FF7C
```

The type of the pointer has to match the type of the variable being pointed to:

```
int iValue = 5;
double dValue = 7.0;

int *iPtr = &iValue; // ok
double *dPtr = &dValue; // ok
iPtr = &dValue; // wrong -- int pointer cannot point to the address of a double variable
dPtr = &iValue; // wrong -- double pointer cannot point to the address of an int variable
```

Note that the following is also not legal:

```
int *ptr = 5;
```

This is because pointers can only hold addresses, and the integer literal 5 does not have a memory address. If you try this, the compiler will tell you it cannot convert an integer to an integer pointer.

C++ will also not allow you to directly assign literal memory addresses to a pointer:

```
double *dPtr = 0012FF7C; // not okay, treated as assigning an integer literal
```

The address-of operator returns a pointer

It’s worth noting that the address-of operator (&) doesn’t return the address of its operand as a literal. Instead, it returns a pointer containing the address of the operand, whose type is derived from the argument (e.g. taking the address of an int will return the address in an int pointer).

We can see this in the following example:

```
#include <iostream>

int main()
{
    int x(4);
    std::cout << typeid(&x).name();

    return 0;
}
```

On Visual Studio 2013, this printed:

```
int *
```

This pointer can then be printed or assigned as desired.

Dereferencing pointers

Once we have a pointer variable pointing at something, the other common thing to do with it is dereference the pointer to get the value of what it's pointing at. A dereferenced pointer evaluates to the *contents* of the address it is pointing to.

```
int value = 5;
std::cout << &value; // prints address of value
std::cout << value; // prints contents of value

int *ptr = &value; // ptr points to value
std::cout << ptr; // prints address held in ptr, which is &value
std::cout << *ptr; // dereference ptr (get the value that ptr is pointing to)
```

The above program prints:

```
0012FF7C
5
0012FF7C
5
```

This is why pointers must have a type. Without a type, a pointer wouldn't know how to interpret the contents it was pointing to when it was dereferenced. It's also why the type of the pointer and the variable address it's being assigned to must match. If they did not, when the pointer was dereferenced, it would misinterpret the bits as a different type.

Once assigned, a pointer value can be reassigned to another value:

```
int value1 = 5;
int value2 = 7;

int *ptr;

ptr = &value1; // ptr points to value1
std::cout << *ptr; // prints 5

ptr = &value2; // ptr now points to value2
std::cout << *ptr; // prints 7
```

When the address of variable value is assigned to ptr, the following are true:

- ptr is the same as &value
- *ptr is treated the same as value

Because *ptr is treated the same as value, you can assign values to it just as if it were variable value! The following program prints 7:

```
int value = 5;
int *ptr = &value; // ptr points to value

*ptr = 7; // *ptr is the same as value, which is assigned 7
std::cout << value; // prints 7
```

A warning about dereferencing invalid pointers

Pointers in C++ are inherently unsafe, and improper pointer usage is one of the best ways to crash your application.

When a pointer is dereferenced, the application attempts to go to the memory location that is stored in the pointer and retrieve the contents of memory. For security reasons, modern operating systems sandbox applications to prevent them from improperly interacting with other applications, and to protect the stability of the operating system itself. If an application tries to access a memory location not allocated to it by the operating system, the operating system may shut down the application.

The following program illustrates this, and will probably crash when you run it (go ahead, try it, you won’t harm your machine):

```
#include <iostream>

void foo(int *&p)
{
}

int main()
{
    int *p; // Create an uninitialized pointer (that points to garbage)
    foo(p); // Trick compiler into thinking we're going to assign this a valid value

    std::cout << *p; // Dereference the garbage pointer

    return 0;
}
```

The size of pointers

The size of a pointer is dependent upon the architecture the executable is compiled for -- a 32-bit executable uses 32-bit memory addresses -- consequently, a pointer on a 32-bit machine is 32 bits (4 bytes). With a 64-bit executable, a pointer would be 64 bits (8 bytes). Note that this is true regardless of what is being pointed to:

```
char *chPtr; // chars are 1 byte
int *iPtr; // ints are usually 4 bytes
struct Something
{
    int nX, nY, nZ;
};
Something *somethingPtr; // Something is probably 12 bytes

std::cout << sizeof(chPtr) << std::endl; // prints 4
std::cout << sizeof(iPtr) << std::endl; // prints 4
std::cout << sizeof(somethingPtr) << std::endl; // prints 4
```

As you can see, the size of the pointer is always the same. This is because a pointer is just a memory address, and the number of bits needed to access a memory address on a given machine is always constant.

What good are pointers?

At this point, pointers may seem a little silly, academic, or obtuse. Why use a pointer if we can just use the original variable?

It turns out that pointers are useful in many different cases:

1) Arrays are implemented using pointers. Pointers can be used to iterate through an array (as an alternative to array

indices) (covered in lesson 6.8).
2) They are the only way you can dynamically allocate memory in C++ (covered in lesson 6.9). This is by far the most common use case for pointers.
3) They can be used to pass a large amount of data to a function in a way that doesn't involve copying the data, which is inefficient (covered in lesson 7.4)
4) They can be used to pass a function as a parameter to another function (covered in lesson 7.8).
5) They can be used to achieve polymorphism when dealing with inheritance (covered in lesson 12.1).
6) They can be used to have one struct/class point at another struct/class, to form a chain. This is useful in some more advanced data structures, such as linked lists and trees.

Don't worry if you don't understand what most of these are yet. Now that you understand what pointers are, we can start taking an in-depth look at the various cases in which they're useful, which we'll do in subsequent lessons.

Conclusion

Pointers are variables that hold a memory address. They can be dereferenced using the dereference operator (*) to retrieve the value at the address they are holding. Dereferencing a garbage pointer may crash your application.

Best practice: When declaring a pointer variable, put the asterisk next to the variable name.
Best practice: When declaring a function, put the asterisk of a pointer return value next to the type.

Quiz

1) What values does this program print? Assume a short is 2 bytes, and a 32-bit machine.

```
short value = 7; // &value = 0012FF60
short otherValue = 3; // &otherValue = 0012FF54

short *ptr = &value;

cout << &value << endl;
cout << value << endl;
cout << ptr << endl;
cout << *ptr << endl;
cout << endl;

*ptr = 9;

cout << &value << endl;
cout << value << endl;
cout << ptr << endl;
cout << *ptr  << endl;
cout << endl;

ptr = &otherValue;

cout << &otherValue << endl;
cout << otherValue << endl;
cout << ptr << endl;
cout << *ptr << endl;
cout << endl;

cout << sizeof(ptr) << endl;
cout << sizeof(*ptr) << endl;
```

Quiz solutions

1) **Show Solution**

6.7a -- Null pointers

Index

6.6 -- C-style strings

Share this:

Email | Facebook 23 | Twitter | Google | Pinterest

C++ TUTORIAL | PRINT THIS POST

41 comments to 6.7 — Introduction to pointers

Skylark
February 3, 2008 at 3:17 am · Reply

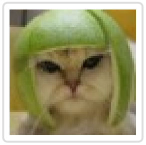
gets headache

Well, that was almost fun to read!

What exactly is the POINT of a pointer? So far I'm just thinking of pointers as duplicates of other variables with similar functionality.

You can determine the address of a value a pointer is 'pointing' to, but can't you just use the '&' operator to do that anyway?

I don't understand whyyyy! T^T



Alex
February 3, 2008 at 11:18 am · Reply

Setting a pointer to point at local non-array variable is an easy way to introduce pointers, but it's not done all that often in practice.

Pointers are used for a lot of things:
1) They are the only way you can dynamically allocate memory in C++. This is by far their most common use.

This topic is covered in lesson 6.9.
2) You can use them to step through the values in an array (as an alternative to array indices).
3) You can use them to pass a large struct/class to a function in a way that doesn't involve copying the entire struct/class, which is inefficient (covered in lesson 7.4)
4) You can use them to pass a function as a parameter to another function (covered in lesson 7.8).
5) You can use them to achieve polymorphism when dealing with inheritance (covered in lesson 12.1).

All of these things are covered in future lessons, but you've got to start somewhere. And this is where. 🙂

DanL
February 29, 2016 at 4:14 pm · Reply

Imagine you want to send an array with ten-thousand elements to a function. If you send the whole array, memory has to be allocated and all of the elements copied. If you send the pointer to the function instead, the function just gets the address of the array, not a new copy of the whole damn thing. If the function makes changes to the array, they are global (kind of), not just changes to a copy. It's a way for disparate portions of a program to work on big bodies of data without needing a ton of global variables and a bunch of copies of the data.

Sakthi Sai Saranyan
November 4, 2008 at 12:55 am · Reply

Very good tutorial. At last I could understand Pointers. 🙂

Darren Fuller
November 16, 2008 at 2:08 am · Reply

Thank you, for the first time I've read something about pointers and it all makes sense and surprisingly easy going compared to other texts I've read.

chkwa
January 24, 2009 at 7:53 am · Reply

if pointer holds the address of a variable, what if i want to know the pointer's address it self how can i do that?

Alex
February 5, 2009 at 11:55 pm · Reply

You can use the & operator to get the address of any variable (even a pointer).
eg.
int nX = 3; // nX has address 0x0012ff60. nX has value 3.
int *pnX = &nX; // pnX has address 0012ff54. pnX has value 0x0012ff60
cout << pnX; // prints value of pnX, which is 0x0012ff60 cout << &pnX; // prints address of pnX which is 0x0012ff54

xian
September 1, 2010 at 9:56 pm · Reply

This is a wonderful tutorial - the examples here clearly step through all possible pointer/pointed-to configurations. Thanks so much!

SWEngineer
January 22, 2011 at 8:29 am · Reply

Simple well explained tutorial.

Thanks a lot.

capitanui
June 7, 2011 at 4:28 am · Reply

I have a question …i cannot understand something.
I have this simple code :

```
int *pValue;
*pValue = 4;
cout<<*pValue;
```

It's all ok..it prints 4;

I i declare another one my program crushes and i cannot understand why since is the same thing.

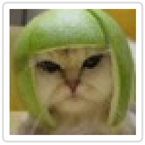
```
int *pValue, *nValue;

*pValue = 4;
*nValue = 5;

cout<<*pValue<<*nValue;
```

zingmars
June 7, 2011 at 4:00 pm · Reply

Something must be wrong, because the first code in which you declare only one pointer crashes too.

Shreevardhan
June 8, 2011 at 1:22 am · Reply

No memory allocated. That is why it crashes.



Alex
August 12, 2015 at 1:00 pm · Reply

Here’s what’s actually happening: pValue was never initialized. Therefore, pValue is pointing to garbage. When you try to dereference a garbage pointer, you get undefined behavior, which often manifests as an application crash.

Ollie999
June 22, 2012 at 7:02 am · Reply

on my computer (windows 7 64-bit) sizeof(pntr) returns 4 when I was expecting it to return 8 based on this tutorial. Does anyone know why this is?

Ollie999
June 22, 2012 at 7:13 am · Reply

Also my computer has 8gb of ram so if pointers are only 32-bit for some reason then surely only half the computers memory could be used? I don’t get it.

Alex
August 12, 2015 at 1:05 pm · Reply

Even though your computer and OS are 64-bit, you’ve defined your project as creating a win32 (32-bit) application. This means your pointers will be 32-bits, and you’ll only be able to address up to 4GB of memory.

If you changed your project settings to compile as a 64-bit application, you’d see your pointers become 8 bytes in size.

KILLDOZER
December 28, 2013 at 12:25 am · Reply

Hey, Alex. I just wanted to let you know I really appreciate you not only giving away all this knowledge without asking in return, but doing so in such a well structured way. In my book, this makes you an admirable person. On a sideline, with the knowledge I’ve acquired thus far, I started making a text-based dungeon crawler. I’ve already made functions which can generate a random map, and draw it using X’s & whitespaces. I’m confident that it will turn out fairly nicely, and it’s all thanks to you.

alexlydiate
June 26, 2014 at 8:19 am · Reply

This is a beautifully clear explanation - many thanks.

Joseph
February 8, 2015 at 8:26 pm · Reply

For being simple, clear, well-explained and easy to understand, I still don’t get pointers.
I get that they have purpose and are extremely useful in large applications, I just can’t wrap my head around the way they’re used.
Just seeing a pointer makes me super confused and a little angry that I can’t understand them as easily as everyone else does.

I’ll try going through this section a few more times to see if it helps.

Alex
August 12, 2015 at 1:29 pm · Reply

For other readers in the same boat, read the next few articles (especially **6.9 -- Dynamic memory allocation with new and delete**). Seeing more context for how they are actually used in non-trivial ways should help it click.

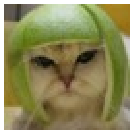
Julie
March 31, 2015 at 9:43 pm · Reply

I think I got it. hope its application will be easy 😀

```
int nValue = 5; // nValue has address 0x28fefc. nValue has value 5.
int *pnValue = &nValue; //pnValue has address 0x28fef8. pnValue has value 0x28fefc

cout << nValue <<endl; // prints value of nValue, which is 5.
cout << pnValue <<endl; // prints value of pnValue, which is 0x28fefc
cout << *pnValue <<endl; // prints value of *pnValue, which is 5
cout << &pnValue <<endl; // prints value of &pnValue, which is 0x28fef8
cout << sizeof(nValue) <<endl; // prints value of sizeof(nValue), which is 4
```

Robot Cat
April 15, 2015 at 11:17 pm · Reply

Hello,
Can you tell me What different between :
- Pointer to a deallocation memory
- Null Poiter (Pointer assign = 0)
- And Pointer undefined (declaration without assignment)
Or they are similar ?



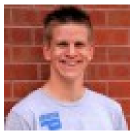
Alex
August 12, 2015 at 1:34 pm · Reply

By "pointer to a deallocation memory" I presume you mean a pointer that is pointing to memory that has been deleted. These pointers will have a valid memory address, but the memory they are pointed to is no longer allocated for the application's use.

In the second case, the pointer is set to null. This is a way of indicating that the pointer is not pointing at anything.

In the third case, an uninitialized pointer hold a garbage address from whatever was in memory beforehand. If you're lucky, this will end up being zero'd out bits, so the pointer will be treated as a null pointer. If you're unlucky, this will be to some random memory address not allocated for your program's use.

Dereferencing any of these pointers will lead to undefined behavior -- likely an application crash.



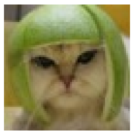
Todd
July 7, 2015 at 12:34 pm · Reply

Typo.

"In other words, when pnPtr is assigned to &nValue:"

I think you mean:

"In other words, when &nValue is assigned to pnPtr:"



Alex
August 12, 2015 at 1:36 pm · Reply

Fixed. Thanks!

Shivam Tripathi
July 23, 2015 at 12:25 am · Reply

I have a confusion in Pointers declaration….Is an asterisk (*) important to place before a pointer variable or any simple variable which when assigned an address of another variable can also become a pointer…i mean to say that consider this:

```
int nVar=7;
int* nPtr=&nVar; //this is a pointer variable holding the memory address of "nVar"
```

But suppose i write lyk this:

```
int nVar=7;
int nPtr=&nVar; //will this "nPtr" also become a pointer OR THE ASTERISK (*) HAS GOT IT'S IM
PORTANCE IN
                // DECLARING A POINTER?
```

I ran this sample in my Dev-C++ compiler and got the expected O/P…it states that “invalid conversion from int* to int”…I understand this but why this conversion doesn’t happen..infact memory address is actually a number only…Even i tried with “double” in order to increase the storing range

And Alex…plz respond to my question being asked in section 6.3 (Arrays & Loops)…seek ur advice…plz..:-)

Alex
July 23, 2015 at 1:51 pm · Reply

I think you answered your own question. Yes, you need the asterisk to denote that the variable is a pointer. If you don’t have it, C++ will complain about an invalid type conversion. The whole point of a strongly typed language (like C++ is) is to make sure you don’t inadvertently make conversions that don’t make sense, such as assigning a memory address to an integer rather than an integer pointer.

liyuan
August 2, 2015 at 1:58 am · Reply

Hi Alex,

I’m wondering why you name your variables with a small n in front, such as in "nValue". I think it’s (or i read it as) (integer)(Value) but I’m not 100% sure since you name short variables as nValue too.

Alex
August 2, 2015 at 7:42 am · Reply

Ignore the prefixes. I used to recommend a certain prefix-based naming convention, but best practice has moved away from such, so I’m in the process of removing all the prefixes as I rewrite the lessons.

Joe
September 1, 2015 at 8:06 pm · Reply

Ok, this is killing me… what is this line?

void foo(int *&p)
{
}

foo takes as input a dereferenced address of p???

Also, this site is amazing, thank you!

Alex
September 2, 2015 at 6:20 pm · Reply

In this case, p is a reference to the integer pointer argument. We haven’t covered references yet. But the short of it is that references can modify the argument values passed in. So when we call foo(p), the value of p may be changed by foo(). The compiler doesn’t know at compile-time whether foo() will assign a valid value to p or not, so it allows us to bypass the error we’d otherwise get about trying to reference an uninitialized value.

Manoj Kumar
November 8, 2015 at 10:03 am · Reply

```
#include<iostream>
using namespace std;
int main()
{
    const int num_rows=10;
    const int num_cols=10;
    int product[num_rows][num_cols]={0};
    {
    for(int rows=0;rows<num_rows;++rows)
    {
        for(int cols=0;cols<num_cols;++cols)
        product[rows][cols]=rows*cols;
    }
    for(int rows=1;rows<num_rows;++rows)
    {
        for(int cols=1;cols<num_cols;++cols)
        cout<<product[rows][cols]<<"\t";
        cout<<"\n";
    }
    return 0;
    }
}
```

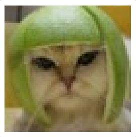
Jim
November 24, 2015 at 10:08 am · Reply

Alex,
You should mention that all pointers are stored in the stack.

Under Assigning a value to a pointer:

You wrote "One of the most common things to do with pointers is have them hold the address of another variable."

Don’t you mean a variable, not another variable? Another make this sound like there are two variables being pointer at.



Alex
November 25, 2015 at 1:49 pm · Reply

Pointers aren’t necessarily stored on the stack. It’s possible to dynamically allocate pointers. When that happens, the dynamically allocated pointers get allocated on the heap.

I used the word “another” to mean a different variable, rather than the address of itself (having a pointer hold its own address isn’t very useful). I’ve updated the word “another” to “a different”.

Jim
November 24, 2015 at 2:06 pm · Reply

Alex,

I mistook what you wrote below to be related to the code above it. May I suggest that you add the word "below" after "Note". So others don’t make the same mistake that I did.

"Note that when the address of variable value is assigned to ptr:
•ptr is the same as &value
•*ptr is treated the same as value"

Rob G.
November 29, 2015 at 12:28 pm · Reply

In my humble opinion much of the struggle with pointers initially is dealing with confusion generated by the same items serving separate purposes. For example
in the following code *nptr seems to be a single item just like a variable. Yet it has different purposes depending on context:

```
int nVar=7;
int *nptr=&nVar // *nptr is initialized to store address of nVar. (&nVar)
cout<<*nptr;    // *nptr is there again, but this time to dereference
```

Alex has done a really good job with this introduction. Thanks Alex!

DanL
February 29, 2016 at 4:18 pm · Reply

In the section "The address-of operator returns a pointer," if you read it one way, you imply that &x returns A POINTER to the address of x (or a pointer to a pointer to x), rather than the address of x. I suggest re-wording it a bit. For clarification: &x evaluates to a hexadecimal address of x, and a pointer getting that value would be a pointer to x. Variables do not have unique pointers. There can be multiple pointers to x.

Alex
March 6, 2016 at 1:22 pm · Reply

&x does not evaluate to the address of x, it evaluates to a pointer that holds the address of x. That pointer can be evaluated to get the address of x. It is done this way because the pointer contains type information, whereas an address literal would not.

Shiva
March 13, 2016 at 7:18 am · Reply

Typo: in the Declaring a pointer example, you wrote:

```
int *iPtr; // a pointer to an integer value (should be 'variable' instead of 'value', right?)
double *dPtr; // a pointer to a double value (here too)
```

Another thing: in more than one example you mixed up assignment with initialisation. I mean:

```
int *ptr = &value; // assign address of variable value to ptr
```

should technically be:

```
int *ptr = &value; // initialise ptr with the address of variable value
```

Or, to specifically show assignment (as in the topic 'Assigning a value to a pointer'), wouldn't it be better to write:

```
int *ptr;
ptr = &value; // assign address of variable value to ptr
```

Suggestion: it'd be good if you could include a note on how the & operator can be used to get the address of even a pointer variable itself in the lesson. Got it from your answer to reader chkwa's question above. I don't know how useful this is practically, but it clarifies a good deal about the concept.

Nice quiz! Sums up everything taught in this lesson perfectly. And no matter how much I tried, the invalid pointer example runs fine on my system without crashing. Duh.

Alex

March 14, 2016 at 5:04 pm · Reply

```
int *iPtr; // a pointer to an integer value (should be ‘variable’ instead of ‘value’, right?)
```

No, a pointer doesn't have to point to a named variable, it can point to a memory address that contains a value.

Fixed the assignment vs initialization comment error. Thanks!

Taking the address of a pointer variable is pretty uncommon. Although you can do it, in most cases you won't have a reason to.

Shiva

March 16, 2016 at 8:12 am · Reply

I didn't know that about pointers. Thanks! 🙂

If I did take the address of say an int pointer, and store it in another int pointer, what is the latter's type info? Is it int **, or int * itself?