The Great Pointer Conspiracy

One of the great tragedies of C and C++ is that they are taught wrong — that a number of perfectly straightforward features are taught and described as if they were mythical and supernatural entities that no mortal can truly understand. Memory management in C++ is one such feature (it is actually very simple, once you know the trick), but the biggest of all is probably pointers.

Everyone who learns C++ fears pointers. *Everyone* who is new to the language, or who has merely *heard* of the language consider pointers to be some kind of magic — arcane constructs that give the programmer access to *Real Ultimate Power* — a feature that both mark C/C++ as *superior* and *more powerful* than other languages, but is also feared as *dangerous* or unsafe*.

None of this is true.

Pointers are simple.

Pointers are not magical.

Pointers are safe (as long as you use them only as allowed by the language)

It is very well-defined what you may, and may not, do with a pointer. The only problem is that the compiler is unable to enforce most of this, so it relies on your own discipline, and knowledge of the rules. But the rules exist. And if you stay within the rules, if your C++ program is legal, then pointers are perfectly safe.

This post is my little attempt to debunk The Great Pointer Conspiracy. It seems there is some hidden rule that whenever we teach others C or C++, we must describe pointers

- as more complicated than they are, and,
- **as something they are not**. It sometimes makes sense to lie to your pupil in order to teach them the truth a bit at a time (similar to how most of what you learned in elementary school turns out to be wrong when you get to university. They didn't mislead you, they just taught you simplified versions of the truth to get you on the right track). But in the case of pointers, the model taught is not merely wrong, it is also more complex and harder to understand!

So what is a pointer then?

Let's start with a crash course in syntax, just to get that out of the way.

- A pointer to type T is denoted `T*` (pronounced *pointer to T*)
- A pointer is created with the `&` operator. Assuming an `int i`, we can create a pointer to it: `int* p = &i;` (`&i` is typically pronounced as *take the address of i*)
- A pointer can be *dereferenced* with the `*` operator, yielding the value it points to: `int j = *p;`

That's easy, right? The only point of confusion is the dual role of `*`, as both part of the type, and as the dereferencing operator. There's a bit of symmetry here, because `&` can be used in both places as well. As above, it can be used to take the address of an object, but it can also be used as part of the type, to create a *reference*: `int& k = i` creates a reference to the previously defined integer i. But references aren't the subject of this post. I only mention it because of the related syntax.

So, on to what pointers are, and what they can do:

Pointers are references

A pointer is little more than a reference (in the conceptual sense — not the specific C++ references mentioned in the previous section) to a variable. If we have multiple references to the same variable, they will all see changes made by each others. Here's an example:

```
void Foo(int* ptr){ // Because we're passed a pointer, we have a reference to the original variable, and can modify it so the changes are vis
  *ptr = 2; // set whatever ptr points to, to 2
}

int main(){
  // create a local variable i. This isn't a pointer, but it can be referenced by one.
  int i;
  int* p = &i; // create a pointer to i by taking the address (see below) of i, and store that as a pointer p
  i = 1;
  assert(*p == 1); // the value referenced by p is now equal to 1
  Foo(p);
  assert(i == 2 && *p == 2);
}
```

important note: Yes, I used the word "address" in the comment above. It is important to realize what I mean by this. I do *not* mean "the memory address at which the data is physically stored", but simply an abstract "whatever we need in order to locate the value. The address of `i` might be anything, but once we have it, we can always find and modify `i`. If you want a real-world analogy, what is an address in the real world? My email-address has nothing to do with my house address. My phone number could be considered a third address. Even my social security number, or my full name could be considered addresses in this sense. All of these allow you to locate or contact me, which is all we require.

So far, so good. Pointers are simply references to other variables, with slightly quirky syntax in that we have to use `*p` to get the value that the pointer `p` points to, and we have to use `&i` to create a pointer to `i`.

Of course pointers can do a bit more than this though. They're not as complex as people often try to convince beginners, but they're not *that* simple either.

```
int main() {
  int i = 1;
  int j = 2;
  int* p = &i; // make the pointer p point to i
  assert(*p == 1);
  p = &j; // and now make it point to j
  assert(*p == 2);
  *p = 3; // modify the variable p points to
  assert(j == 3);  // j is now 3
  assert(i == 1);  // but i is untouched, because p no longer points to it.
}
```

See, that's not rocket science either, is it? Whatever the pointer points to, we can look at and modify. And when it no longer points to that, they have no connection any more.

Pointers can be null

Next up, pointers don't have to point to something. They can be *null pointers*. And just like with addresses in the previous example, it is important to be clear on what we mean by this. A null pointer is exactly what I said: *a pointer which does not point to any object*.

In particular, it is *not* a pointer to the address zero. Of course, here is where it becomes tricky, because the following *does* create a null pointer:

```
int* ptr = 0;
```

The trick here is that the C++ language standard makes a special rule for this case. Assigning the constant zero to a pointer creates a null pointer, and *not* a pointer to address zero. The "constant" part is important too. Here is the precise wording in the standard (Section 4.10 [conv.ptr], paragraph 1:

> A *null pointer constant* is an integral constant expression (5.19) rvalue of integer type that evaluates to zero. A null pointer constant can be converted to a pointer type; the result is the *null pointer value* of that type…

A "constant expression" is essentially an integral value which can be evaluated at compile-time. So `42`, `2+2` or `const int i = 99` are constant expressions.

```
int* p0 = 0; // null pointer
const int zero1 = 0; // constant expression
int* p1 = zero1; // null pointer
const int zero2 = 2 - 2; // constant expression
int* p2 = zero2; // null pointer
int zero3 = 0; // not a constant expression
int* p3 = zero3; // not a null pointer
int a = 2;
int b = 2;
int zero4 = a - b; // not a constant expression
int* p4 = zero4; // not a null pointer
const int c = 2;
const int d = 2;
int zero4 = c -d; // constant expression
int* p4 = zero4; // null pointer
```

Obviously, the compiler is unable to enforce all of this, but that doesn't make it less true. According to "the rules", a null pointer is neither a pointer pointing to address zero, or a pointer to which the value zero has been assigned. It is *a pointer to which the constant expression zero has been assigned*.

As for what you're allowed to do with a null pointer? Basically nothing. You may compare it to other pointers, and… that's basically it.

With me so far? You might have noticed that what I have described so far is almost exactly what references in C# or Java (or many other languages) are. A variable of a reference type behaves pretty much exactly like this. We can set it to point to another *valid* object (but we are *not* allowed to ever set it to an *invalid* object), or we can set it to `null`.

Pointers are much like reference types in most other languages. This is an important point. Like I said to begin with, pointers are *not* difficult. They are a very simple concept, as the above shows. Where the confusion arises is in the *one* extra thing they can do, which I will describe next. Note that while this *does* make them somewhat more flexible than C# references, it is still a far cry from the "raw memory address" concept that people often think pointers are.

Pointers can traverse arrays

Now comes the (slightly) tricky part — the one that usually gets people confused, or gives them the wrong idea. If we have a pointer to an element within an array, we are allowed to move the pointer around within the array

```
char arr[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j' };
```

```
char* ptr = arr; // arrays are not pointers, but can *decay* into a pointer to the first element. So now we have a pointer to arr[0]
assert(*ptr == 'a');
++ptr; // move ptr to the next element
assert(*ptr == 'b');
ptr += 5;
assert(*ptr == g);
assert(*(ptr + 2) == 'i');
assert(*(--ptr) == 'f');
ptr -= 3;
assert(*ptr == 'c')
```

So far, so good. At this point it is probably a good idea to mention that when you increment a pointer, it always moves *to the next element*, and *not to the next byte*. Once again, being careful with the idea of "addresses" pays off. The pointer stores the address of an object. By adding one to that address, we get the address of the *next* object, no matter how big the object is. Think of your house address. It doesn't matter how big your house is, the *next* address is always the neighboring house. It is not your garage door or your kitchen window.

If, for the sake of argument, pointers had merely been memory addresses, then adding one to a pointer would have produced an address that was one byte higher, which means the pointer would no longer have pointed to a valid object. Good thing we don't live in *that* messy kind of world, eh? In C++ land, a pointer points to an object, and

```
char* p = arr + 9;
assert(*p == 'j'); // no surprises here, just verifying that we're at the end of the array.
char* q = arr + 10; // this is legal
++p; // so is this
```

But once again, we have to be careful. The language has only given us permission to go *one* step past the end. A pointer to `arr + 11` is downright illegal, *even if we don't dereference it. The mere existence of the pointer is illegal*. The compiler probably won't complain, and your code may even *appear* to work, but it is no longer a legal C++ program.

We have also not been given permission to dereference the one-past-the-end-pointer. `*(arr + 10)` is not legal. Again, it may seem to work, on your computer, with your compiler, on this particular day. But it may not work tomorrow. Or on my compiler. Or when I run your program.

So the language allows us to create, and move pointers around freely, from the start of the array, and up to one past the end of the array. And it allows us to dereference pointers that point to any element in the array, but not one past the end.

And that's basically it. This is the dreaded pointer arithmetic that usually have beginners running scared. Not all that scary, is it?

Of course, For the sake of completeness, there is one other arithmetic operation that is legal under much the same circumstances:

Two pointers *pointing to the same array* may be subtracted, yielding the distance between them, expressed as a number of elements. And for the purposes of pointer arithmetics, single elements are considered arrays of size one, meaning that all the above is true for single variables too — they're just treated as arrays with only a single element.

And one final detail

Now let's get self-referential. There is nothing new in this — it follows as a logical conclusion of the above, but it often comes as a surprise, so let's mention it:

Pointers may point to pointers. Again, there is no magic, no special cases. A pointer is simply a reference to an object, remember? And a pointer is an object too, so obviously we can point to *that* as well!

We don't often need to do that, but there is one case where it is used. Typically when you call a library function, and want it to give you a pointer to some resource it has created, you do this:

```
Resource* ptr = 0; // this is going to be our pointer to the resource. For now, make it a null pointer to avoid confusion
bool success = CreateResource(&ptr); // pass the address of our pointer to the function
```

Note that the function wishes to return a status code to let us know if the operation succeeded, so it can't simply return the pointer we want. So it has to resort to pointer-pointer trickery instead.

The insides of `CreateResource` might look something like this:

```
bool CreateResource(Resource** res){
  Resource* actualResource = new Resource(); // create the resource, and temporarily store a pointer to it
  // now we need to pass this pointer to the caller. If res had been a regular "single" pointer, it would simply have been a null pointer.
  // And sure, we could have made it point to our resource instead, but the caller wouldn't know, because we only received a *copy* of the or
  // Instead, we use a pointer to a pointer. We know that 'res' now points to the caller's Resource pointer. So if we manipulate the value po
  *res = actualResource; // so take our newly allocated resource pointer, and store that into the caller's pointer, which we get by dereferen
}
```

It may help to remember that function arguments in C++ are *always* copied. If you pass an `int` to a function, it receives a *copy* of that `int`. And if you pass a pointer, then the function receives a *copy* of that pointer. A copy which points to the same address, so anything we do to the pointed-at address will be visible outside the function as well. But if we change the pointer itself, no one else will see it, because the function has been given its own copy.

So if we pass a pointer `p0` to a pointer `p1`, then this is again copied. The function receives a copy of `p0`, let's call it `p2` which points to `p1`. So if we change what `p2` points to, the calling function won't see it, but if we change what `p1` points to, it will be visible to the caller, because `p0` still points to `p1`.

Yes, this added level of indirection may take some getting used to, but the important part is that there's nothing fundamentally special. It is simply the logical conclusions of the rules I described previously, so even if you don't get it now, you will when you've got a bit more experience with pointers. It's similar to how, when you first learned to read "See Spot Run", you had all the rules necessary to read longer words, like "stewardesses" or "programmatically". After that, you pretty much just needed practice.

So that's it. That's all pointers are. If you hadn't previously encountered pointers, you can stop reading here. But if you were already taught about pointers, we probably have to undo some of the damage.

So the following will discuss what pointers are *not* — that is, the misconceptions that typically exist about pointers, and which beginners are almost invariably taught. I'll try to explain *why* these limitations exist as well, partly so you can take the rule seriously as "something with real-world relevance".

The Pointer Abuse Rehab and Correction Center

In the following, assume that `i, j` are integer variables (`int`), and `p, q` are pointers to integers (`int*`) and `n` is a null pointer:

- A pointer is not just a number. For example, `i + j` is legal, but `p + q` is not. Try it. Your compiler will give you an error. Likewise, `i*j` is valid, but `i * p` is not. Integers may be added to or subtracted from pointers, and pointers may be subtracted from pointers (as long as they both point to the same array). And on some computers, a pointer isn't implemented as an integer either. Some machines have segmented memory space, so an address is a tuple consisting of a segment identifier plus an offset. Sure, you *can* combine those two in a single number, in the same way that you can combine the country code with my phone number to create a single integer. But the address is still, fundamentally, a tuple of two numbers on that machine.
- A pointer is not a memory address! I mentioned this above, but let's say it again. Pointers are typically *implemented* by the compiler simply as memory addresses, yes, but they don't have to be. A pointer may not point to just any address (and again, some computers, which have separate address and integer registers, are actually able to enforce this at runtime, generating a hardware fault if you try to create a pointer to an address that is not allocated to your process.) The same goes for

- All pointers are not born equal. A pointer to T may not be convertible to a valid pointer to U. Some machines require datatypes to be aligned. Typically, a 4-byte integer will have to be aligned so it starts on an address that is divisible by 4. But a single byte datatype such as a char can be placed anywhere. So that means three out of four char pointers will not be valid integer pointers! We also can't rely on casting as much as we'd typically expect. `reinterpret_cast` in particular often trips people up. (For non-C++ programmers, you can assume that we had used the "traditional" casting syntax, as in `(float*)i`. The difference is not important.)

```
int* i; // assume we have a pointer i and that it points to a valid integer
float* f = reinterpret_cast<float*>(i); // #1
int* j = reinterpret_cast<int*>(f); // #2
assert(i == j);
```

In the above, we know *nothing* about the value of `f` after the cast on line `#1`. We know that it contains an "implementation-defined mapping" of the original `i`. But we are *not* guaranteed that it points to the same address, *or even that it contains the same bit pattern*!

True, the standard says that the mapping is "intended to be unsurprising to those who know the addressing structure of the underlying machine", but in general, we can't rely on that. All we are guaranteed is that once we cast *back* to the original type, we're given the original value. So the standard guarantees that `i` and `j` in the above will point to the same address. But we know nothing about `f`, other than that the compiler is able to convert the value stored in it back to the original pointer `i`.

Conclusion

By now, I hope it's clear that pointers actually become a lot simpler when we treat them as what they are, reseatable references to objects. If we start pretending that they are memory addresses, we get a whole host of complications: we start thinking that they should be allowed to point to *any* memory address, or even worse, that they are just numbers, and that all the usual arithmetics work on them. (Remember, adding or subtracting integers is legal, but it adjusts the pointer by that number of *objects*, not *bytes*, as we would have expected if pointers were just memory addresses. And `pointer + pointer`, `pointer * pointer` or `pointer / pointer` are simply not defined at all.)

As if that wasn't bad enough, we also require the student to understand the underlying hardware, in particular the concept of a memory space, and of physical (or virtual) hardware addresses.

But if we treat pointers as what they are, that is no longer necessary. A pointer points to a C++ object, not a memory address, so to understand pointers you merely have to understand C++ objects, not memory addresses.

Tags: c++, pointers, teaching

4 Responses to The Great Pointer Conspiracy

1. *Bernd Jendrissek* says:
   April 28, 2013 at 21:28

   You refer to the "dual role of *, as both part of the type, and as the dereferencing operator". It actually isn't a dual role at all — it's one and the same! In a pointer declaration, that * is just another operator that's allowed in a type declaration — along with [] and () (the function call operator). Think of the right-hand side of a declaration (not just pointer declarations!) as a specification of what one has to do to the variable to yield an atom of the type specified by the left-hand side.

2. *jalf* says:
   April 28, 2013 at 22:42

   I don't see how you get to that conclusion. They're related, sure, but they're not "one and the same". An operator applied to a value is not, and can never be, the same as a part of a type specification. When used in a type declaration, it is not an operator. Neither are [] and (). Syntactically they're the same, but not semantically. (Again, there's obviously a *relationship* between those two semantic meanings, in that they are both used to relate a pointer to the type it points to. But hardly "one and the same") :)

3. *uggs* says:
   August 6, 2014 at 14:16

   10 , Mercer a dansé il a manière juste au-dessus de duc .,ugg homme Les principaux avait été folle en semences juste . Quiconque qui monitored l' aventure est conscient Mercer était un étudiant en gérer de graines de fruits . Cependant, en dehors de l' particulier Possède

4. *Rolling audio Rack* says:
   July 2, 2015 at 10:57

   Jonathan Budd states that with this product one will be a part of the "Inner Circle". Plus the boat speakers should be able to absorb the shock of bouncing waves and not damage the quality of the sound. One other suggestion that I could give to you is to try to brand your business whenever possible.

Leave a Reply

Name and Email Address are required fields. Your email will not be published or shared with third parties.

Website

Submit Comment

- Find
- Pages
  - About
  - The DikuSTM library
- Categories
  - Games (6)
  - Meanwhile (12)
  - Meta (6)
  - Programming (31)

- Archives
  - January 2012
  - August 2011
  - July 2011
  - June 2011
  - May 2011
  - April 2011
  - March 2011
  - January 2011
  - December 2010
  - November 2010
  - September 2010
  - August 2010
  - July 2010
  - June 2010
  - April 2010
  - March 2010
  - February 2010
  - January 2010
  - December 2009
  - November 2009
  - October 2009
  - September 2009
  - August 2009
  - July 2009

- About
- The DikuSTM library