Objects and Pointers in C++
Scott D. Anderson

C++ handles objects and pointers a little differently than Java. First of all, pointers in Java are relatively invisible, but in C/C++, they are "first class" things (you can explicitly make them and pass them around).

Why is this useful? One major reason is that pointers are lightweight things, while many of the things we use in databases are big, heavy things. For example, a "page" in our databse is 512 bytes; other database pages might easily be 8K or more. You don't want to have multiple copies of those in memory, you don't want to have to move them around (copy them from place to place), and you don't want to be constantly creating and destroying them. Pointers, on the other hand, are just a single word (4 bytes), which is literally the easiest and most efficient quantity to copy around. Indeed, one of the objectives of this assignment is to ensure that *all* of the pages that are in memory are managed by the buffer manager, so all clients of the `BufMgr` will use *pointers* to pages, *never* allocating a page themselves.

1 Pointers and Addresses

Here's how to make an integer variable and how to make a pointer to an integer:

```
int x;
int* px;
```

Notice the asterisk (star) that indicates that `px` points to an integer.

We all know how to make `x` have a value, what about `px`? A pointer is an address, so we want to give `px` the address of an integer. We have one right there, so let's take the address of it. For that, we need the C "address of" operator, which is an ampersand. Think about the following example:

```
px = &x;
*px = 5;                // x=5
*px = *px + 1;          // x=x+1
if( x == 6 ) return true;
```

- The first line sets `px` to be the address of `x`, so it points to the same memory location that `x` corresponds to. (Think back to your assembly language class: it's code like this that makes C be described as "high level assembly language.")
- The second line "dereferences" the pointer: it says to make the thing pointed to by `px` equal to 5. In other words, it's entirely equivalent to the code in the comment.
- The third line just reinforces the point about dereferencing: the `*px` on the RHS of the statement means "the thing that `px` points to," and since `px` is declared to point to an integer, that starred expression is an integer (in this case, 5).
- The fourth line will be true in any implementation of C/C++.

2 Pointers, Addresses and Objects

Rarely will we want to point to integers; that was just for warm-up. Integers, after all, are small like pointers, so there's no problem with copying them around. The fact that the preceding code is possible does not mean anyone advocates it. Blot it from your memory. Done? Good.

In real life, we point to objects. Our next example is how a linked list is done in C/C++. I will call each "link" in the chain a `Link` and a linked list is just a series of these things in which each points to the next in the sequence. Here's how one could construct a linked list of three integers.

```
class Link {
public:
   int data;
   Link* next;
}
```

Notice the datatype of the `next` field: it's a pointer to a link. It *can't* be a link, otherwise links would contain themselves. There are lots of examples of recursion in computer science, but in every case, you have to be able to refer to something without *being* the something. A linked list is a recursive data structure, but the actual object cannot contain itself. (What size would it have?)

Next, the code to construct a list of length three:

```
Link* make_list123() {
   Link* a = new Link;
   Link* b = new Link;
   Link* c = new Link;
   a->data = 1;
   b->data = 2;
   c->data = 3;
   a->next = b;
   b->next = c;
   c->next = NULL;
   return a;
}
```

It's helpful to draw this with boxes and arrows; I don't have time to do that in this handout, but please do so. Here, I'll give you some room for that:

Okay, let's talk about some of the new syntax. Creating a link is just what you'd expect from your knowledge of Java, except for the star, because `New` is defined to always return a *pointer* to the allocated memory.

Next, there's this funny `->` notation. That is used when we have a pointer to an object and that object has named parts. So `a->data` is the `data` field of the object pointed to by `a`. You can use that just like any other expression.

Here's a dirty secret. I'm going to tell you just to reinforce your intuition, but I want you to promise to forget it. Okay? Remember how in Java you say `x.data` if `x` is a `Link` object and we want the `data` component? The *same* thing works in C/C++! However, `a` isn't a link, it's a pointer to a link. How can we dereference a pointer to the thing that `a` is pointing to? By using the star! So, the following two expressions are, by *definition*, equivalent in C/C++:

```
a->data = 3;
(*a).data = 3;
```

In other words, the "arrow" operator is defined to dereference the pointer and then take a component of the object that it was pointed to. However, every C/C++ programmer uses the arrow operator, not the stars and dots, so you should now erase this from your memory. Thanks.

In fact, in your program, almost everything should be a pointer to an object, so you should find yourself almost always using the arrow syntax and almost never using the dot syntax. Just create pointers to objects and use an arrow wherever you would use a dot in Java.

Let's look some more at the linked list idea. We need to be able to follow along a linked list to find something or delete something or whatever. Here's a bit of code that searches for `t` in the linked list returned by the previous function:

```
bool find_t(int t) {
   Link* list = make_list123(); // points to first element of list
   Link* tmp = list;            // points to the same place
   while(tmp != NULL ) {
      if( tmp->data == t ) return true;
      tmp=tmp->next;
   }
   return false;
}
```

This is the classic algorithm to run down a linked list. Notice that at the end of this code, `tmp` will either point to the link that contains `t` or to `NULL`, but fortunately, we still have the variable `list`, which points to the beginning of it. (This example code will lose the list, but your code will avoid that problem.)

A Problem-Oriented Example

How would you use the ideas of pointers, addresses and objects in this assignment? Your buffer manager will have an array of `Page` objects and clients will want to be able to use one. For example, when you pin a page, you'll want to return to the user a *pointer* to the page frame that they can use. (Not the whole page; that's too bulky.) Consider the following code fragment:

```
class BufMgr {
private:
    Page frames[100];           // Costs us 51200 bytes!
    ...
public:
    Status pinPage(PageId_t page_num, Page*& page_ptr);
    ...
}
```

The `frames` variable is 100 page objects. The `pinPage` method (which is just declared here, not defined), returns a `Status`, which will be `OK` if all is well or, say, `NO_FRAMES_LEFT` is something prevents it from pinning. The `pinPage` takes an argument of the page number (called `pnum`), which is of a special type called `PageId_t`, which is just a synonym for an unsigned int. The other argument, `page_ptr` will be *set* by the method, so it's an "output" argument.

Now, here is a snippet of how `pinPage` might work.

```
BufMgr::pinPage(PageId_t page_num, Page*& page_ptr) {
   ...
   // found a frame, its number is fnum
   ...
   // read the page from disk and put it in frames[fnum]
   ...
   // ready to give the client a pointer to it:
   page_ptr = &frames[fnum];
   return OK;
}
```

So, the buffer manager has 100 page frames, which is just an array of `Page` objects, and the client is given a pointer to one of them, the one in frame `fnum` (a number from 0–99). In fact, the client won't even know *which* frame it has, nor will it care! What will the client code look like? Here's a snippet:

```
void user_code() {
   BufMgr* BM = new BufMgr(100);
   Page* my_page;    // this does NOT create a page
   Status s;
   ...
   // pin page 0.
   s = BM->pinPage(0,my_page);
   if( s != OK ) ...
   ...
   my_page->setData("this string is written on page 0");
   ...
   // unpin this dirty page
   s = BM->unpinPage(0,true);
   if( s != OK ) ...
   ...
}
```

The first line creates a buffer manager with 100 page frames, using the constructor method for the `BufMgr` class.

You can see that the client code doesn't have to allocate any space for the page object—just a pointer. The call to the `pinPage` method sets the pointer, and we can use it later. (The need to set the pointer, by the way, is why the argument has an ampersand: it needs a reference to the argument so that it can change it, because C/C++ is a call-by-value language.) In fact, if we were to use the pointer before it is set to point to the page (say by moving the `setData` method before the pinPage), we will get a "segmentation fault." Segmentation faults are almost always caused by uninitialized pointers or following pointers that point to the wrong kind of thing.

You'll notice another syntax that may be new to you: the arrow with a method invocation. In C++, as in Java, if you have an object, you can invoke a method on it by the syntax `obj.meth()`. But, as we saw earlier, C++ uses the arrow syntax when we have a pointer to an object, so `ptr_obj->meth()` is how to invoke a method on the object that the variable points to. Notice that that's how we invoke the pinPage method and the setData method. In your assignment, you'll define the pinPage method, but I've already defined the setData method for you, so look at the code and see how that works.

3 Memory Allocation

In Java, all memory (except for a few built-in objects, such as ints) is dynamically allocated from a mass of storage called the "heap." In C/C++, for reasons of efficiency, you have a choice of statically or dynamically allocating memory from either the stack or the heap. I won't get into the details of the difference now—that would be covered in a programming languages class—but I will talk about how to dynamically allocate memory from the heap in C++.

3.1 Static Allocation

First, we need to talk about static allocation, and become familiar with three arrays. Then we'll see how to do them dynamically. Consider the following definitions:

```
class BufMgr {
   Page frames[100];
   int queue[100];
   Link* hashtable[100];
}
```

Each of these definitions will cause an array of 100 things to be created. (It's not actually static, but the difference isn't important right now.) The first is an array of 100 objects, each of which is a whole Page, so there are no pointers there at all. The second is an array of 100 ints, so again there are no pointers and everything is straightforward. The third is an array of pointers (pointers to Links, to be exact). However in the last case, it's important to remember that the syntax doesn't create *any* links; it just creates a bunch of uninitialized pointers to such things, should we ever have any to point to.

The downside of this style of allocation is that we have to know in advance (at compile time, hence statically) exactly how many things there will be in each array. We would like to be able to decide that at run time.

3.2 Dynamic Allocation

The reasoning behind dynamic allocation is based on another dirty secret of C/C++, and this is one that you don't have to forget, at least not right away: the datatype of an array is definitionally equivalent to a pointer to its first element. You may remember from assembly language that an array is just a sequence of memory locations, and you can calculate the address of any element just by knowing the address of the first and the size of each one. That's *exactly* what C/C++ does, and so an array is equivalent to a pointer to its first element.

Why is that fact, which seems like a trivial implementation detail, interesting? Because it explains how we can define the datatype of an array when we *don't* know how big it is. It's declared as a pointer to the datatype of its element type. Let's re-do the declarations of `BufMgr`, above, using this idea:

```
class BufMgr {
   Page* frames;
   int* queue;
   Link** hashtable;
}
```

Note that in each case, we deleted the bracketed number of elements, and put a star at the end of the datatype. The first variable, `frames` is a pointer to a Page (later, when we allocate space for the array, the variable *will* point to the first element). The second variable, similarly, points to an integer. The third, holy moly, has *two* stars! That's because it's an array of pointers to links, so a pointer to the first element must be a pointer to a pointer. Yes, that's weird and seems complicated, but if you think of it just syntactically, you can see that we still just deleted the bracketed number of elements and added a star to the datatype; it's just that this time the datatype already had one star, so now it has two.

Okay, now that we have pointers to non-existent arrays, how do we actually do the dynamic allocation? Here's a possible constructor method for the `BufMgr` class. The constructor (just like the one we saw in `user_code`, above) has an argument that specifies the number of frames (we're calling it `NF`).

```
BufMgr::BufMgr(int NF) {
   frames = new Page[NF];
   queue = new int[NF];
   hashtable = new Link*[NF];
   // init hashtable elements to NULL
   for(int i=0; i<NF; i++) hashtable[i] = NULL;
   ...
}
```

The first three lines all have the same syntax: the variable is set to a new array (dynamically allocated from the heap by `new`) of `NF` elements, each of datatype specified to the left of the brackets. The third one has `Link*` in it, because each element is, in fact, a pointer to a link.

Note that while you do have to have exactly `NF` frames in your buffer manager, you don't have to have exactly `NF` elements in your queue or hashtable, though that's not an unreasonable amount, either. It's the one I used. I can expand on this more, later, if you'd like. It's part of what makes hashtables so very cool. Basically, the expected (or average) length of any bucket is the number of things that you'll put in the hashtable divided by the number of buckets. In this case, we'll put up to `NF` frames in the hashtable, and so if we have `NF` buckets, the expected list length will be NF/NF or 1. That means that our time complexity to look something up in the hashtable will be $O(1)$. So, we get the time-complexity of an array and fast insertions as with a linked list. Isn't that cool?

4 Conclusion

Why is C/C++ so weird and complicated in this way, and why am I requiring that our coding projects be in C++? Here are some reasons:

- Efficiency: one of the primary reasons that people turn to C/C++ is when they feel that efficiency is paramount. Because C is so "low-level," it becomes much easier to know exactly when something is efficient. Also, C and C++ allow stack-allocation of memory, which I didn't discuss, but can be very useful in improving efficiency. Java has many excellent features, but speed and efficiency are not among them.
- Size/Data representation: In Java, the exact representation of objects—the components and how they are laid out in memory—is hidden from the programmer (for good reason, see below), but one of the things we must grapple with in databases is the layout of data in memory. Regardless of our exact page size, 512 or 8K, it has a strict limit, and we need to be able to control how things are laid out. C/C++ allow that. We didn't see that here, but things like fixed-size arrays, laid out as contiguous memory locations, and the difference between a pointer to a thing and the thing itself, are crucial.
- Recklessness: The designers of C/C++ believe in speed over safety (unlike Java's designers), so they give the programmer access to and control over pointers, addresses and all kinds of "low-level" things. In estimates of the frequency of different kinds of bugs, pointer bugs always rank high: think about that the next time Word or Netscape crash. Java aims to be "pointer-safe," which reduces efficiency and requires pointers and addresses to be hidden from the programmers, but reduces crashes. C/C++, on the other hand, have "segmentation faults."
- Access to OS calls like random-access file operations. There undoubtedly are equivalent operations in Java, but C has been around for thirty years longer than Java, and those operations have been well tested.
- popularity: C/C++ are the most widely used languages (not necessarily the best) and it would be a big hole in your education not to at least have seen them a little and tried them.
- I'm a sadist: I chuckle maniacally when I see students juggling chain saws.

Seriously, there are some new ideas here, worth drawing pictures and thinking about. They're very cool ideas, in some cases, and worth expanding your mind for. You may ultimately reject C/C++ as an awful kludge—and you wouldn't be the first—but get to know it a little before you do so.