Here's some notes about common points of confusion about the differences between Java and C++. I CANNOT stress enough that it would be valuable for you to draw diagrams showing how (your abstract view of) memory progresses as programs execute. This will GREATLY improve your ability to understand code!

I've put space between questions and answers so you can answer them yourself first.

In Java, everything that is not primitive is a pointer. ("Reference" means pointer.)

So, for example, consider the following Java code:

```
  class Node {
    public int data;
    public Node next;  // line A
  }

  ...

    Node p;            // line B
    p = new Node();    // line C
    p.data = 5;        // line D


    Node q;            // line E
    q = p;             // line F
    q.data = 1;        // line G
```

What's the type of the variable p declared on line B of the code?

(A: NOT a Node. It's "a reference to a Node".)

Does line B create an actual Node (i.e., a Node object)?

(A: No, it doesn't.)

What's the type of q from line E?

(A: A reference to a Node.)

Does line F create a new Node object?

(A: No. q and p both refer to the same Node object.)

What's the value of p.data before line G?

(A: 5.)

What's the value of p.data AFTER line G?

(A: 1.)

When is the new Node created on line C collected for reuse?

(A: sometime after there are no more references to it in the program; could be FAR from this code!)

Now, consider the same code in C++:

```
  struct Node {   // a struct is the same as a class
                  // except members default to public
    int data;
    Node next;         // line A
  }

  ...

    Node p;            // line B
    p = new Node();    // line C
    p.data = 5;        // line D


    Node q;            // line E
    q = p;             // line F
    q.data = 1;        // line G
```

This will fail to compile. Why?

First of all, the thing to be uncertain of in C++ is NOT pointers. A variable of type Node* in C++ (which is the type of "a pointer to a Node") is JUST LIKE a variable of type Node in Java. However, a Node in C++ is a new beast.

In both Java and C++, "new Node()" returns a pointer to freshly created Node object. In Java, that's cool because p is of type "pointer to a Node" and so can store the returned pointer.

What's p's type in C++?

(A: A Node or, if you want to emphasize: a Node object.)

Do the types match?

(A: No, they don't. A pointer to a Node and a Node are NOT the same things.)

Now, let's revisit the other questions above:

Does line B create an actual Node (i.e., a Node object)?

(A: YES, it does. p IS a Node.)

Well, if B creates a Node, we don't even NEED the "new Node()" call
because we already have a Node. So, let's patch the C++ code:

```
  struct Node {   // a struct is the same as a class
                  // except members default to public
    int data;
    Node next;        // line A
  }

  ...

    Node p;            // line B/C
    p.data = 5;        // line D


    Node q;            // line E
    q = p;             // line F
    q.data = 1;        // line G
```

Now it compiles. Let's revisit some of the other questions:

Does line F create a new Node object?

(A: No; q and p are each already Node objects.)

What's the value of p.data before line G?

(A: 5.)

What's the value of p.data AFTER line G?

(A: 5. WHY? What happened on line F? Did it make p and q both point to the same object? It CANNOT have, because they are NOT POINTERS!)

When is the new Node created on line C collected for reuse?

(A: as soon as the variable p goes out of scope. It WILL NOT exist after this function call completes and CANNOT be used by later code!)

Why will this still not compile?

(A: A Node that contains a Node inside of it (the next variable) must take infinite memory. That doesn't work! So, we need to change the declaration of next to "Node* next;".)

Finally, let's change the Nodes to Node*s. In that case:

```
  struct Node {   // a struct is the same as a class
                  // except members default to public
    int data;
    Node* next;        // line A
  }

  ...

    Node* p;           // line B
```

```
    p = new Node();    // line C
    p->data = 5;       // line D


    Node* q;           // line E
    q = p;             // line F
    q->data = 1;       // line G
```

Now, ask the same questions as in the Java case. Do any of them differ?

(A: Generally no. The one key difference is that that new Node will NOT be deallocated until you explicitly deallocate it. You could do that, in this code, with either of:

```
  delete p;
```

or:

```
  delete q;
```

Note that neither of these statements affects p or q in any way.

Instead, it deallocates the memory that p (or q) refers to.)

Finally, a quick summary. If you're a Java programmer, then:

(1) You will likely be MORE comfortable with pointers than with non-pointer types (at least for anything that isn't effectively "primitive" like an int). So, wondering whether to declare a variable as a Foo or a Foo*? If you don't have a solid reason for Foo, consider Foo*.

(What's a solid reason? Well, for one: "I'm only ever using this object here in this function and when the function ends, I'm done with it. In that case, use a Foo and the compiler will take care of collecting the memory when you're done.)

(2) The C++ -> operator is EXACTLY equivalent to the Java . operator. The C++ . operator is only ONE PART of Java's . operator. (So, worry more about use of ., not about use of ->.)

(3) Side note: A struct in C++ is nearly exactly the same thing as a class. It's just a class where the members default to public instead of private. Not comfortable with structs? Use a class instead like this:

```
class Foo {
 public:
  ...
};
```

That class is EXACTLY like this struct:

```
struct Foo {
  ...
};
```

(4) If you invoked new, you should probably invoke delete. If you

used brackets [] after new, you should use [] after delete when you invoke delete. For example:

```
  int * a = new int[5];
  ...
  delete [] a;
```

Steve Wolfman
2011/01/21