CPS235 Object Oriented Programming in C++

Pointers and Dynamic Memory

C++ has both pointers and references whereas Java only has references. C++ pointers are manipulated more like Java references. You cannot manipulate (using assignment statements) C++ references at all.

The main difference between C++ pointers and references (in Java or C++) is that the compiler manages references (knows you want the contents, not the address) whereas you (the programmer) must de-reference pointers to get at data. In other words, the compiler automatically does the de-referencing when it's a reference.

To allocate memory, use the keyword *new* . To deallocate memory, use *delete*. There is no garbage collection so all memory allocated by programmer must be deallocated by the programmer. The compiler manages memory it allocates, e.g., int n; The compiler allocates memory for n and deallocates its memory when it goes out of scope.

Let's define some simple **int** pointers or pointers to **ints**. We'll also mix in references and draw pictures for everything. It's essential that you get a correct mental model of pointers. All variables have memory and you need to know exactly what holds what, and whether it's an address or a typed value such as an int.

Define an int n and show its memory
(Every time new code is written, the memory picture is completely redrawn so you can compare the old memory picture to the new memory picture.) The compiler allocates memory for one int. The n label next to it is for us humans. When you access n, the compiler knows and uses its address. The letter n isn't used internally.

```
int n = 10;            // a reference

           n +----+
             | 10 |
             +----+
```

Now literally define a reference
The ampersand is used. You must do the reference assignment on the declaration, i.e.,
`int& ref;   ref=n;` won't compile.

```
int& ref = n;        // Now n is also named ref

           n  +----+
          ref | 10 |
              +----+
```

Setting either n or ref does the same thing. Both assignments access same memory.

```
n = 88;
           n  +----+
          ref | 88 |
              +----+
ref = 99;
           n  +----+
          ref | 99 |
              +----+
```

Now let's define a pointer

The asterisk says the variable is a pointer. Recall that *new* allocates new memory. (The dots in the memory mean it's garbage, some value, but you have no idea what is it.) This is done on two lines of code, separating the declaration and assignment to show memory, but it could be done on one line: int* p = new int(20);

```
int* p;
             +-+
           p |.|
             +-+
p = new int(20);
             +-+     +----+
           p |-----> | 20 |
             +-+     +----+
```

The code int* p; is always read from right to left: p is a pointer to an int . Sometimes in simple cases such as this, for shorthand it is said that p is an int pointer . I would describe this as the int pointer p points to the memory holding the value 20. Points to means holds the memory location, holds an address. Also, the blank in the declaration doesn't matter. The following declarations are equivalent:

```
  int* p;         // C++ people tend to do it this way
   int *p;         // C people tend to do it this way
   int  *  p;    //  rarely  seen  in  code  although  often
seen in error messages
```

The compiler allocates memory for p and the programmer allocates the memory for the int that p points to. The compiler allocates memory at compile time (reserves memory at compile time). The new is dynamic allocation, during run time.

Unfortunately doing it the C++ way can get you into trouble. You might think that

```
   int* p,q;
```

gives you two pointers to ints. But the asterisk really does go with p and this statement is equivalent to

```
   int* p;
   int q;
```

Now derefernce p to set it to a different value

Read *p as the memory that p points to.

```
*p = 40;                  // dereference to access memory p
points to, set to 40
           +-+      +----+
         p |------> | 40 |
           +-+      +----+
```

Define another pointer and do some manipulation. You can visually think of pointer assignment as copying the "arrow" ... of course, you're really copying the address, which the arrow represents.

```
int* q = p;               // q holds the same address as p,
both point to the int 40

           +-+      +----+
         p |------> | 40 |
           +-+      +----+
                    ^
                   /
             +-+  /
           q |----
             +-+
```

Throw the int n from above in the mix. The variable n is of type int. While p and q are pointers, of type int* , *p and *q are of type int.

```
n = *p;                  // n (from above) and *p are both of
type int, n is now 40
           n  +----+
          ref | 40 |
              +----+

*q = 80;                 // q and p point to the same memory
location, now holds 80

           +-+      +----+
         p |------> | 80 |
           +-+      +----+
                    ^
                   /
             +-+  /
           q |----
             +-+

cout << *p << "   " << *q << endl;       // "80   80" is
printed
```

```
cout << n << "   " << ref << endl;       // "40   40" is
printed
```

We would have a memory leak if we terminated now
Allocated memory that p (and q) point to has not been deallocated. If we terminated our program, the compiler would deallocate the memory for p, q, and n, but the dynamically allocated memory that currently holds the value 80 is never deallocated.

This is a major problem for applications such as the operating system because memory that is not deallocated cannot be used by other applications. Eventually you run out of memory. Have you ever wondered why windows starts to run slow after some time? But when you reinstall windows and your applications, it suddenly runs much faster. The answer to the slowdown is MEMORY LEAKS. Don't make them. Keep track of your memory. The rule is ... you allocate it, you must deallocate. Every *new* must have a corresponding d*elete.*

So let's deallocate (delete) our memory.

```
delete p;   // deallocates the memory that p points to
p = NULL;   // for safety, always set your pointers to NULL
```

One other bad thing you can do is a called a dangling pointer
If you deallocate memory that a pointer points to, then it's pointing to garbage. In other words, the pointer contains the address of memory that is no longer yours. Here's an example. Let's start over with our declarations.

```
int* p = new int(50);
int* q = p;

           +-+      +----+
         p |------> | 50 |
           +-+      +----+
                      ^
                     /
             +-+  /
           q |----
             +-+

delete p;
p = NULL;

           +----+  ......
         p |NULL| .      // memory there, but not owned by you anymore
           +----+  ......
                    ^
                   /
             +-+  /
           q |----
             +-+
```

The varialbe q is the dangling pointer. It points at deallocated memory. It may fool you as you could print and it would appear to have a valid value, but since it is not your memory, that memory could be allocated elsewhere and given some other value. The next time you access it, it might be a completely different value. That is always a fun bug to find.

To fix the dangling pointer problem, set the pointer to NULL:

```
q = NULL;
```

Practice drawing memory
Here's all the code from above. Can you end up with the same pictures?

```
int n = 10;          // n is a reference
int& ref = n;        // n is also named ref
n = 88;
ref = 99;
int* p;
p = new int(20);
*p = 40;              // dereference to access memory p points to
int* q = p;           // q holds the same address as p
n = *p;               // n and *p are both of type int
*q = 80;              // q and p point to the same memory location
delete p;             // deallocates the memory that p points to
p = NULL;             // for safety, always set your pointers to NULL
q = NULL;
```

Mix ints, pointers, and references one more time
You can also make pointers point to ints although since you did not allocate the member, you cannot deallocate the memory. Starting a new example, consider the code:

```
int n = 30;            // n is a reference
int& ref = n;          // n is also named ref
int* p = &n;          // can get to n via p
```

In the last line of code, the &n is read as the address of n, or the reference of n. The first two lines show ref as an alias for n as seen before:

```
            +----+
         n  |    |
       ref  | 30 |
            +----+
```

Throw the pointer p into the mix and you get the following picture:

```
                   n +----+
          +-+     ref|    |
        p |------>   | 30 |
          +-+        +----+
```

Now you can get at the exact same memory in three different ways. The following three assignments all accomplish the same thing, to set the int memory to 10.

```
   n = 10;
   ref = 10;
   *p = 10;
```

Taken from: http://courses.washington.edu/css342/zander/css332/pointers.html