**Michael’s TOP SECRET lesson plan for Week 6 “pointers” topic.**

Pointers are variables that store a memory location.

That’s it.

They are “just variables” like anything else (int, string, etc.).

While pointing to a general memory location *might* be useful, it’s much more useful to be pointing to a “specific type of thing”, so a pointer (memory location) to an “int” would be “different” than a pointer to a “string”.

HOWEVER: they are actually NOT different – they are just storing a memory location.

Remember: EVERYTHING is just numbers to the computer, so a memory location of an “int” or a “string” or even a function, are *just* memory locations.

Since this is the OPPOSITE of helpful, you make pointers specific to the type it’s pointing to.

This is really so the *compiler* that knows what’s going on, not that it will prevent you from doing something “stupid”.

For instance:

* You can “reinterpret\_cast” which allows you to force one type to another.
* You can use a “pointer to void” (void\*) which allows you to point to anything.

To specify a pointer, you add a “\*” after the type:

* int x ; // an int
* int\* px; // a “pointer to an int” (the “p” is for pointer, but it can be any variable name

To get the memory address of something, you use the “address of” operator, or “&”.  
(Which is the same as “by reference”, which is a little confusing)

int x = 5;  
int\* px = &x; // px has the address of x, or “points to” x

We normally don’t want the actual address, but what it’s “pointing to”, so:

cout << px; // Prints the memory address (not too useful)

cout << \*px; // Prints the thing that the pointer is pointing to  
 // If x = 5, then this would print out “5”

This is called “de-referencing” the pointer, getting at what it’s pointing to. You can treat a de-referenced pointer just like the original variable, so \*px = 8 would set x = 8 (the thing px is pointing to)

While this is interesting, it’s not super useful. In standard “C”, there was no “by reference” (it’s a C++ thing), so they way that you would “pass by reference” was to “pass by pointer, like this:

void DoitLikeCPP( int &x ) // C++ way ‘by reference’

void DoitLikeC( int\* px ) // C way ‘by pointer’

Inside the function, the C “pass by pointer” would have to “de-reference” the pointer to manipulate it:

void DoitLikeCPP( int &x )  
{  
 x = 5;  
}

void DoitLikeC( int\* px )  
{  
 \*px = 5; // Note the de-reference  
}

For us, in INFO1156, we are more interested in pointers as they relate to the control of when objects are created and deleted and if we only want ONE instance of an object, rather than “copies” of the object.

cPerson\* pBob = new cPerson(); // Creates person, pointed to by pBob.

We get at the data with the clunky “de-reference”:

(\*pBob).firstname = “Bob”

...or with the “indirection” or “arrow” operator:

pBob->firstname = “Bob”

If you have a vector of pointers, rather than regular “stack” objects, then they are pointing to the same thing. In other words, there’s only one of the thing they are pointing to, rather than copies.

vector< cPerson\* > vec\_pPeople;

cPerson\* pBob = new cPerson;

vec\_pPeople.push\_back(pBob); // Saves a *pointer to* the “bob” person, not a copy/clone

pBob->firstname = “Bob”;

cout << vec\_pPeople[0]->firstname; // prints out “Bob” (vector and pBob point to the same object)

Lastly, you have to delete the objects when you are done;

delete pBob;

Note that if you delete the object in one place, invalid pointers can still be hanging around, like the pointers in vec\_pPeople are still pointing to where “bob” *was*, even though it’s been deleted.

So be careful of that.