Week 3, Day 2 – Lesson plan:

Review:

* Arrays in C
  + At compile time
  + At run-time (using new)
    - Watch out for the delete
  + Are technically “pointers”
  + Can’t know the size (number of elements), so:
    - Need to keep this around as well
  + Can’t change size
  + Any index is “valid” (to the compiler), even if it cause a crash
* std::vector
  + Is templated, so my specify type
    - GENERATES SPECIFIC CODE *FOR THAT TYPE*
      * Created AT COMPILE TIME
      * They have NOTHING TO DO WITH EACH OTHER
    - ALL templated code is like this
    - Is ONLY useful when code is “generic” aka: you KNOW you are going to use a different type, but want to reuse THIS code.
    - You will learn more template stuff in 3rd term
      * It should *not* be your “go to”
      * It is *not* “better”
      * It’s not even necessarily easier to follow (often the opposite)
  + Is a full blown class, so has useful things like:
    - .size() returns the number of elements (things) it’s go in it
    - Is zero size to start
    - Behaves just like an array (because there’s literally an array in it)
      * Implements the “operator[]()” (aka “[]”) index
        + Means “get the thing at location [n]”
        + Does *not* test for out of bounds

Just like an array

* + - * Note: This is not explicitly stated in the standard, but the easiest and most sensible implementation is an array, so that’s what they use
      * C++11 standard makes this even more explicit with the .data member
    - Add items with .push\_back(), which is a strange name, but it’s that it “pushes” an item onto the “back” of the array:
      * Adds the item at the “end” (actually at the .end() location)
      * Increases the index value
  + Sets aside a certain amount of space
    - NOTE: There ISN’T anything there at first, tho!!
  + When it hits that “limit”, it grows, usually doubling:
    - Allocate ANOTHER block of memory
    - Copy old data to new
    - Delete old block
    - This is slow, and gets slower
  + Can use .reserve() if you want to avoid this “re-grow+copy” issue
  + Can NEVER get smaller (by design)
    - Can use vector<type>(myVector).swap(myVector); // Scott Meyers Effective STL
      * Uses “copy constructor” to re-create vector, copying only the amount needed. It’s a copy
    - C++ added . shrink\_to\_fit
  + Be careful with returning – vectors can be large
    - Can be a performance killer
* Iterators:
  + A “short hand” to locations inside a “container”
  + Build in ones: .begin() and .end()
  + Syntax is like a pointer: “\*” symbol “dereferences” the iterator
    - i.e. the iterator “points to” the thing
      * ...but *isn’t* “the thing”
    - You “dereference” to get the actual thing
  + SIDE NOTE:
    - STL is ALWAYS “copy in, copy out”
      * i.e. what’s in the container is a COPY of the thing
      * what’s returned is a COPY of the thing
    - Be careful when returning or using “=” because IT WILL COPY
  + You can also use iterators to point to whatever.
    - If it’s a “random access” iterator:
      * It behaves like an index in an array/vector
      * Use ++ and –
      * Move it with number (like += 3)
      * Can set it to something
    - Note all iterators are “random access”
      * Usually depends on the container (eg: lists *aren’t* random access)
  + For loop with iterator
    - ISN’T “faster” or “better” (or even “clearer”, necessarily)
    - But it’s likely fine
  + So why use them?
* The algorithm library (aka: why you bother to learn iterators):
  + It’s big. We’ll look at just some key ones.
  + Today, we’ll look at sort. That’s it.
  + Let’s sort a bunch of numbers (yawn... #lame)
  + How does it “know how to sort?”
    - Hint: does “<”, “>”, “==”, “!=”, etc. make sense with built in/native number types?
    - Hmmm.... yeah, the compiler “knows” how to compare these!!
  + What about types it doesn’t “know?”
    - i.e. everything you’re actually supposed to deal with
  + Let’s make a bubble sort:
    - Let’s look at the key thing in the bubble sort: the compare
      * It ALWAYS compares TWO THINGS
      * **ALL** sorts do this, not matter how “bougie”/sexy/fancy they are
      * Why?
        + Because computers can only compare two things.

#truestory #dontbelivethehype

* + - If we have a “compare these two things”, we can:
      * Change the “compare” part and compare whatever we want
      * Swap out the sorting technique (algorithm), too!
    - “Predicate function”
      * Think: How to compare these two things?
      * Pass this function
      * Tada!
    - There’s also a “functor” (predicate function object)
      * operator() is overloaded
      * Just like the predicate function, but this signature
      * It’s because () is rarely overloaded