INFO-1156 Week 3, Day 1 – Michael’s Lesson Plan:

|  |  |  |
| --- | --- | --- |
| 2: January 19th | * Data types and built in structure  (brief intro: arrays, structs, classes – more later) |  |
| 3: January 26th | * I/O (input/output), console and files * Functions * Command line arguments (“C style”/stack arrays) | Assignment/Quiz #1 (2.5%) Quiz (think: short, in class) |

NOTE:

* These are MY notes, of “what I need to go over in class”
* Think of this as “what Feeney might write on the white board”
* This is a point form thing, that I’m explaining as I go, so it’s a reminder
* If it helps you to look at this, then great. If not, then don’t look at it.
* This is the notes (you would hope!) your instructor has for each lesson.
  + Fun game: see if you can guess which instructors have these and which don’t!

Review:

* Types
* Functions, signatures
* Header (.h) vs Implementation (.cpp) files
  + Extern
  + Declaration vs assignment
* Compile and link (“build” process)
  + Compile error
  + Link error
  + “Library” (Debug, Release, etc.)
  + “Architecture” (x86, x64, ARM, etc.)
  + There is also the “run-time” which is often tied into the compiler
    - The “VS 2019 run-time” vs the “VS 2013 run-time”
* C/C++ programs run “natively” on the hardware
  + There is no virtualization (no virtual machine like in Java, C#) or interpretation, like in Python, PHP, Lua, etc.
* File input and output
  + How did you do reading that census file?
  + How did you handle the commas?
    - There is not “best way”.
    - Any library is literally scanning through the things character by character.

Casting:

* “C style” cast: (type), like “(int)x;” casts to an “int”
* C++ equivalent: static\_cast<type>, so: “static\_cast<int>(x);”
  + Other casts:
    - dynamic\_cast: used for inheritance
    - const\_cast: “un-consts” something
    - reinterpret\_cast: force compiler to do it
      * “never” use this. It’s like using a sledgehammer on a wood screw.
      * It’s useful in very rare cases, but it usually means bad code.
  + They aren’t “better”, but may give more clarity for your intention
  + Casting can only deal with types that *can* be cast
    - You can’t “cast” a string as an int, for example.
    - You can *convert* a string to an int (maybe).
    - With casting, thing “can be treated as”
      * An int “can be treated as” a float
      * This works with inheritance, too:
        + A cSuperNinja “can be treated as a” cNinja, for instance

Basic struct, class

* Identical, but:
  + Struct is public by default.
  + Struct suggests only data, not methods
    - i.e. if a struct has methods (beyond c’tor), then it’s “huh??? But why?”
  + Class suggest methods
    - Similarly, a class *without* methods might get a “um...” but not as much

Arrays:

* Compile time (stack). Can’t change size. Can’t even know size.
* Run time (heap, aka “new”).
  + Side branch into malloc and free
    - Are “C” memory allocation
    - Some systems just wrap “new” around malloc
    - (you will see this, but don’t use them)
    - It’s still often very important to know how much actual memory something is.
      * i.e. malloc/free use “number by bytes” ignoring the type
        + Same thing that “sizeof()” does
      * new/delete handle “number of things of that type”
* Command line arguments:
  + Is a “C style array”, because it’s, well, “C”
  + Note that it also passes the size of the array
    - If it “knows” how big the array is, why would it do this, huh?
      * Because it *doesn’t,* that’s why.
  + Can convert to a string if you’d like

Vector (with the STL)

* Based (loosely) on “smart arrays” (like Java, but certainly not unique to Java)
* Is a template class so you have to specify the type you want
  + VERY IMPORTANT: templates are “blueprints” for generating code, so:
    - SPECIFIC CODE IS GENERATED FOR EACH TYPE YOU MAKE:
      * Vector<int> 🡪 makes a vector that handles ints
      * Vector<float> 🡪 makes a COMPLETELY DIFFERENT vector that handles floats
      * Vector<cHelloKitty> 🡪 makes another COMPLETELY DIFFERENT vector that handles cHelloKitty types
      * They have ABSOLUTELY NOTHING TO DO WITH EACH OTHER; they just “sort of look the same” because they were *generated* by the same process
    - This also means:
      * Errors happen in the *template* code, not *your* code (usually)
      * It’s an error of “I don’t know how to make this work” from the compiler
      * Templates are a “find and replace, then generate *this* code” process, which will help track down the error
    - But they are very powerful
* Have zero size to start (nothing in there)
  + You “add” to the “end” of the vector, making indices as you go
    - Call “push\_back()” to “push” the item to the “back” (end) of the vector, adding an element
  + Never shrinks. Seriously, they don’t. You have to do this yourself.
  + There is a “size()” method that returns... wait for it... the size
    - Is a “size\_type” which is almost always an unsigned int, but is “whatever type makes sense to that container”. So you might get a compiler warning.
    - You can keep it “size\_type”, cast it to an int/unsigned int, or just ignore the warning (level 3; level 4, which is bone headed, will assume this is an *error* which it is not.)
  + Your an “reserve()” space to speed up pushing back data.
    - This does *not* create the thing you are storing, just allocates some space.
      * Think: a limit on your credit card: if you increase it, it doesn’t mean you have more money, right?
  + “clear()” clears the items, but *doesn’t* actually shrink the vector (still takes the same size in memory), which is usually fine.
  + Vector is supposed to “look, taste, and smell” line an array, so:
    - “[]” is defined (which is actually the operator()[]() method) where you pass an index (as size\_type/unsigned int) and it returns the “thing” at that location.
      * It will behave *just like* an array if it’s out of bounds
    - “at()” does the same thing, but changes how the exceptions are made
      * Side note: don’t use exceptions for programmatic control: they are for *error conditions*. This is for many reasons, including that many C/C++ projects have exceptions turned OFF for performance.
        + Exceptions kill performance (“unwind the stack” and prevent you from predicting how long something will take to run).
        + Also, exceptions don’t handle a lot of things you wish they would (like bad allocation of memory, for instance).
        + This is the *exact opposite* of “Java world”, where exceptions are encouraged, but exceptions in Java (since it’s a virtual machine) are a *very* different thing than they are in C++.
    - Vector also have “iterators” which sort of behave like pointers, and are useful for when we start using algorithms, etc.:
      * Begin() is the start of the vector (the 0th element)
      * End() is *one after* the end of the vector (the nth element)
        + i.e. a vector of 5 items has indexes from 0 to 4
        + begin() would be at [0] and end() would be at 5 (not really, but you can think of it like that)
        + i.e. “end()” is an *invalid* location
      * looping with an iterator
      * “dereferencing” an iterator with the “\*” operator
        + This was taken from pointers (same syntax)