What we did at the Monday, October 30th, 2017 5024 Programming Meeting

(as remembered by your host/mentor, Michael Feeney)

* We reviewed some things:
  + Every standard C/C++ program has one, and only one “main()” function.
  + We need at least one cpp file, called anything we want (i.e. *doesn’t* have to be called main)
  + The standard main() function can be in two forms:
    - **int main(void)**
    - Note: **int main()** without the “**void**” is valid, too, as the empty brackets “()” imply that there’s a “**void**” inside. “**void**” means “nothing”, so this main() “takes nothing as a parameter”
    - **int main(int argc, char\* argv[])**
    - This main() takes an integer and a “c-style string” (technically, it takes “an array of pointers to the char type” or “a pointer to a pointer of char types)
    - NOTE:
      * *Don’t* worry about the line above or the 2nd form of main() – you can wipe it from your memory if you’d like, but in case you are interested, that’s technically the “other” form of standard main(). If you want to know more about it, feel free to ask, though ☺
      * We *won’t* be using this form of main, anyway. You can still use it, of course, but it’s so that you can type things on the command line. Again, this won’t be used in the robot code, so we won’t investigate it further.
    - Couple comments:
      * There are non-standard mains, too, like a Windows “win32” application (aka a C/C++ application linking to the Windows GUI) uses **WinMain()** instead of **main()**
      * When we get to the actual robot code, there *is* a main, but it’s called for you; it’s actually hidden at the bottom of one of the files. So we won’t write it, is my point.
* The file “extensions” (which are hidden in Windows by default) are *critical* here. The compiler looks at the extension to see what to do with the file.
  + Files ending with “.c” are (usually) compiled using the “C” compiler (which *can’t* compile C++ code...)
  + Files ending with “.cpp” are compiled using the “C++” compiler (which can also compile “C” code, as C++ contains all the C language)
  + Files ending with “.h” are called “header” files, and are *not* compiled by anything.
  + All of these files are simply text files, and are “human readable” (unlike Word, or other files, which aren’t intended to be used outside of their programs).
* “C” and “C++” are not “the same”, but a lot of what we are doing overlaps, so you’ll see “C/C++”, unless it’s specifically talking about one language or the other. “C” is more “low level” and “C++” is (perhaps) more “high level”, but both languages are:
  + “Strongly typed” – meaning you *must explicitly* define the data types (aka variables, and things that deal with variables, like functions, methods, etc.), and “converting”/”conversions” from one type to another (like an int to a float, for instance) follows very specific rules.
  + “Unmanaged” – means that you have to deal with the memory “management” of variables. Other languages like Java, C#, PHP, etc., are “managed”, meaning that they pretend like there’s only one sort of memory, and you don’t have to “clean up” after yourself – i.e. you don’t have to worry about “deleting” variables that you aren’t using anymore. We’ll talk about this more in later weeks.
* “Variables” are:
  + Places in the computer memory to “put stuff”
  + Are of an explicit “type” (a number, a string, a Nijna, whatever, but you have to be clear and specific). Other “weakly typed” languages don’t have this requirement. In C/C++, it gets sort of ridiculous (like an “int” is different from an “unsigned int”, etc., but that’s because it reflects what’s actually happening in the circuitry of the computer. “Weakly typed” languages like PHP just mask/hide this reality from you).
  + Are declared/defined in one place (i.e. what “type” it is) and “initialized” in another. This can happen on the same line, too:
    - int x; // declared or defined (and “allocated”) – x contains random information
    - x = 0; // now x contains 0 (zero)
    - int x = 0; // declared, defined, AND initialized in one line (you should do this)
  + C/C++ is super picky, though conversions are often done automatically, so:
    - 3 // is an integer type
    - 3.0f // is a “floating point” type   
       (technically, an IEEE-745 32-bit long type)
    - 3.0 // is a “double precision floating point type”   
       (technically, an IEEE-745 64-bit or 80-bit type
    - Different types ***cannot***be used together – I’m quite serious: math calculations are done in different areas of the computer, and only on the same basic types: integers with integers, floats with floats, etc.
      * This means that you ***cannot***(technically) “add and integer to a float”. However, you ***can*** convert one type to another, *then* add them.
      * Types can be explicitly converted (called “casting” which we’ll talk about later) or they can be “promoted” (*automatically* converted) to numbers that have a higher range, so:
        + **int x = 3;**
        + **float y = 4.5f;**
        + **cout << x + y;**

What “type” does cout() “see”? And int or a float?

In this case, the x will be “promoted” to a float *just* during the calculation “x+y”, making it a “float+float”, with a result of a float.

So cout(), *in this case,* will “see” a float type.

* + - * It is often not clear what the “best” way to do this is. Often, there *isn’t* a “best” way:
        + Integers are “precise” but have limited “range”
        + Floating point has larger “range” but are not *necessarily* more “precise” and are actually *inaccurate* in many cases (for us with what we’re doing with the robot, they will be good enough, though)
        + The point is that you *don’t* get high precision AND high range with *any* built in types. This is true for almost all languages, unless they have explicit types that handle extended precision+range, beyond the built in “hardware” types (which is pretty rare: like the “Numerics” type in C# (.NET), the floating point value in Ada, and “real” in FORTRAN)
* Functions:
  + Is essentially a way to organize the code – it doesn’t “matter” to the computer at all.
  + They are called “functions” because the code inside them “performs some specific function”
  + Almost always, the function “does something”, and almost always transforming data from one form to another and/or moving data around.
  + Suggestion: Give them a name that “makes sense”, which usually means a verb (action word) and perhaps a noun (person, place, or thing) that involves a data type.
    - Good examples of functions are:
      * **AddNumbers()**
      * **MoveRobot()**
      * **CalculateHST()**
      * ...and so on.
    - Bad functions are names that are unclear as to what they do or are just silly/pointless, like:
      * **Blah()**
      * **Yolo()**
      * **CatsAreInsensitive()**
      * z();
      * ...and so on.
    - Basically, someone should have a sense of what the function is, or what it’s supposed to do, just by looking at the name (and maybe looking at the parameters being passed).
  + Can return a variable.
    - In C/C++, then can only return *one* thing. Now, this can be a big thing (like an array), but can be *only one instance of one type of thing*.
    - The return value is ignored when comparing functions. Really, it’s ignored, so these are *identical:*
      * **void KillAllHumans(void);**
      * **int KillAllHumans(void);**
      * **float KillAllHumans(void);**
  + Can take any number of variables as “parameters” in its “parameter list”. These are passed as “arguments” to the function. The variables are declared just like ‘regular’ variables: type, then name (like “int x;”):
    - **void CalcualteTax( float price, float taxrate );**
    - **void PussycatKillKillKill( std::string dogName, float strength );**
    - **float FindAgeOfStudent(string first, string last, int studentNum );**
  + While we can place functions in the file they are used (like we did at the start), there are many benefits to separating the function into separate files.
    - To do this, we separate the “function implementation” (aka the actual function, like the code that is actually run), and the “function declaration” (also called the “function signature”, which is just the function name and return types (like the three functions listed above... CalculateTax(), etc.)
    - Implementation: can be in *any* cpp file. i.e. the *name* of the file has *no* connection to the function. However, it would be a Good Idea to make the file name “make sense”. Often, the file the function is placed in is related in some way, and there’s almost always *many* related functions in one file, where the file describes the relationship, like:
      * File called **RobotFunctions.cpp**, with functions like: **MoveRobot()**, **GrabBall()**, **KillAllHumans()**. In other words, these functions *together* are related as “things that the robot could do”, aka “robot functions”.
    - A related “header” file (ending in “.h”) WITH HEADER GUARDS is created that holds the function signature(s). Note with ‘regular’ functions (the ones we’ve used), there *isn’t* any actual ‘code’ in the header file; put another way: if you have actual ‘code’, then it *shouldn’t* be in the header file – *only* the signature goes in the header:
    - In the header:

Note the semicolon at the end of the function signature

* + - * **#ifndef \_TaxFunctions\_HG\_  
        #define \_TaxFunctions\_HG\_  
        float CalculateHST( float price );  
        #endif**
    - In the .cpp (“implementation”) file:
      * **float CalculateHST( float price )  
        {   
         return price \* 1.13f;  
        }**
  + A “header guard” is the combination of #ifndef, #define, and #endif that brackets the .h file. Note:
    - The #ifndef should be the 1st thing in the file
    - The matching #endif should be the last thing in the file (i.e. there shouldn’t be anything after the #endif line, other than a blank line)
    - The “token” after the #ifndef and #define has to be *unique in the entire program*. So you *can’t* make it the *exactly* the same as the function, or usually the file name, or any (global) variable used *anywhere* in the program (i.e. in *any* file that’s used in *that* particular program).
    - A common convention is to take the name of the file (maybe put it in ALL CAPS), then add a little flourish at the start and end, like the “\_HG\_” above.
    - Sometimes the header guard “flourish” will have the date the file was created or the name of the person who did it, so something like:
      * **#ifndef \_MagicFunctions\_June\_31\_1980\_  
        ...**  
        or...
      * **#ifndef \_MagicFunctions\_Harry\_Potter\_  
        ...**...for a file called MagicFunctions.h
  + In the file(s) that use the function, you use the #include to “include” or “pound include” (as the “#” was originally called the “pound” key... also, if you say “hash include”, you’ll be mocked. Just saying.)
    - #include “MagicFunctions.h”   
      ...would be in the .cpp (or .h) files that are using the functions listed in the MagicFunctions.h file.
    - Note that *the functions aren’t actually there*, only the “function signatures”...
    - ...but that’s OK, since it’s the “linker” – the thing that runs *after* the compiler compiles all the .cpp files into “object” (.o or .obj files) – and “links” all the function calls to the actual functions.
    - (This is why, even though the compiler only “sees” *individual* .cpp files, it can still “know” how to call other functions – through the function signatures!)
  + The “#” things are technically called “pre-processor directives” and are looked at by.... wait for it... something called the “pre-processor”. This thing reads the files, interprets these # directives, and changes the files before passing them to the compiler. In other words, the compiler never sees the # things.
    - For example #include does a logical copy-and-paste of the file it’s referring to (which is why you need those header guards!)
  + The function *always* returns, but if it doesn’t return anything – i.e. it has **void** as the return type, you often don’t see the return before the bottom closing braces, as it’s implied.

Function short version:

* Put the actual function (the code) into a .cpp file with some sensible name, which can just be the name of the function, but usually it’s the name of a group of related functions (i.e. the file contains many, closely related, functions in it)
* Copy the 1st line of the function (return value, name, and parameter list) into a header file, and add a semi-colon to the end. This is the “function signature”.
* Make sure your header (.h) file has a “header guard”.
* In the files that need the function, #include the *header* (.h) file with the function signature.
  + NOTE: NEVER *EVER* ***EVER EVER NEVER*** #include a **.cpp** file. (***ONLY*** .h files)  
      
    (Technically, you *can* do this, but you can also do other stupid things, too, like eating garbage, jumping off roofs, and playing in traffic, but you *shouldn’t* do these things.)

“Classes”:

* This happened right at the end of the class.
* Many modern languages treat classes like The Way To Do Things, but that’s misleading.
* If you think of header files as collections of *related* functions – i.e. the reason you placed these functions in this common header file was because they are “sort of” related in functionality (**RobotFunctions.h** having **MoveRobot()**, **PickUpBall()**, **ChaseMrTiet()**, **KillAllHumans()**, etc.), a “class” is just a reinforcing of this “related” idea, plus a few more benefits
* Functions are “global” in that they are visible to everything (well, everything that #includes the header file) so:
  + Their names+parameters (aka “signature”) has to be *unique* across the *entire program* (i.e. *all* the files), *even if they aren’t being #included everywhere*. This is because the linker needs to see one, and only one, of each function in the program (think: if you have more than one main() function, like in *different* files, but in the *same* program, which one are you supposed to call?)
  + Sharing data between functions is tricky.
* Another perspective of a “class” is a “type of thing”, but we’ll look at that in a moment. For now, we are thinking of it as “a think that has related functions”
* To create a class, you:
  + Create a header file. Conventionally, this would have identically the same name as the class, but *this is not required at all* in C++. This convention (the file name matches the class) is adopted in languages like Java, but it was borrowed (stolen?) from C++.   
    (You can also have more than one class per file, too, but don’t worry about that for now.)
  + Add a header guard. *Just like with the function* situation, make it related to the file name.
  + Add the following:  
    **class CMyClassName  
    {  
    public:  
      
      
    };**

Your amazing code goes in here... well, your amazing “function signatures” (and soon some data) goes here...

* + ...where **CMyClassName** is the name of your class with a “C” at the start (little or big, doesn’t matter).
    - The “C” is just so you know it’s a *type* rather than a variable.... and it’s just a convention (recommendation). More on that in a moment....
  + Inside the class, you place the function *signatures*. EXACTLY like the functions you know and love!
  + In fact, you can copy and pasts the function signatures into the class! Really, it’s allowed!!
  + So you might have something like this:  
    **class cTaxFunctions  
    {  
    public:  
     float calcHSTOntario( float price );  
     float calcHSTNefoundland( float price );  
     float calcPSTBritishColumbia( float price );  
     float calcPSTManitoba( float price );  
     // ... and so on  
    };**
  + *Just like with functions*, you would place the actual code (now called “methods” – i.e. a “method” is really just a “function” but inside a class) into a .cpp file.
    - Suggestion: make the .cpp the same name as the header (.h) file, which is the same name as the class.
    - So for the cTaxFunctions class, you’d have a cTaxFunctions.h and a cTaxFunctions.cpp file.
    - Add a “class scope operator” to the function(s) in the .cpp file, something like this:
    - **float *cTaxFunctions::*calcHSTOntario( float price )  
      {  
       return price \* 1.13f;  
      }**

I’m a “method” inside the cTaxFunction class! ☺

* + - Note this is almost exactly like regular functions, just that the class name is added:
    - **float calcHSTOntario( float price )  
      {  
       return price \* 1.13f;  
      }**

I’m a “function” that’s **not** tied to any particular class! ☺

* + - So these two things do *exactly the same thing*, in that they have *exactly* the same code, it’s just that one is “inside the class” (we say it is “a method of the class”) where the other is a “function” (technically a “global function”).
  + So why use classes? Good question. Basically, it shows that these “methods” (aka functions inside classes) are VERY closely related (we say they are “tightly coupled” in computer/nerd speak).
* How do you call these methods (functions inside classes)?
  + The “catch” is that a class is really a *type* of thing, like int, float, string, etc.   
    (actually std::string *IS* a class! #mindblown!)  
    so we have to *create* one *as a variable* before we can use its methods.
  + Before, calling **calcHSTOntario()** was a matter of typing it out:  
      
    **float total = calcHSTOntario( 9.99f );**
  + But calling the **calcHSTOntario()** *method* that’s inside the **cTaxFunctions** class means we have to create one of these **cTaxFunctions** first.
  + OK, how do we do that?
  + GOOD NEWS: Just like any other variable!
  + **int x;**  // Creates an “x” variable which is a type of “int”
  + **std::string name;** // Creates a “name” variable which is a type of “string”  
      
    so....
  + **cTaxFunctions myTF;** // Creates a variable “myTF” which is a type of cTaxFunctions!

*It’s almost too easy!*

* Calling the method is similar to calling the function, just that you use the “instance” (i.e. the variable name) and the “dot operator” (i.e. the a period “.”):
* Calling the function:  
    
  **float total = calcHSTOntario( 9.99f );**
* Calling the method:  
    
  **cTaxFunctions myTF;**   
    
  **float total = myTF.calcHSTOntario( 9.99f );**

We will look more into classes as we go. One thing to remember is that classes are really just a way of organizing your code, essentially placing functions into yet another organizing container (in this case, something called a “class”), they *aren’t* some magical, necessary thing.

Classes are, technically, a data type, just like any other type (int, float, string, etc.); they are a “user defined” type.