Comments on Horton for EE342
(About the 2008 edition – 2012 edition may be a bit different.)

Purpose:

We are using Horton, *Beginning Visual C++ Programming* as a reference and textbook for EE342. Time does not permit us to follow through the book systematically, first learning C++ in a console / text context and then adapting that to Windows applications with a "Graphical User Interface" (GUI). Rather, we are diving right into the example code from Freescale / CMX and using Horton as a supplementary reference, looking things up as we need them. This document will say what things in Horton we can safely skip, and comment on some of the others. This document is expected to grow over time as more is added.

Starting Up:

The initial chapters of Horton are a very good place to start for learning C++. We are also programming in C on the microcontrollers, so we have a preference for using forms and constructs that are most similar between C and C++. In other words, I prefer the C way of doing things to the C++ way when there's a choice. And there often is.

Horton describes two basic approaches, actually three, to doing C++ applications. We can write C++ that directly interacts with Widows. OK for a console program maybe, and the most portable to other systems perhaps, but we don't want to do that for a GUI. We can write stuff that uses the Microsoft Foundation Classes. That's what the example code from CMX does. It's why the icon has the letters "MFC" in it – the default icon for such an application. It might be a pain to port such a program to other systems because you'd have to do so for different foundation classes that wouldn't be compatible. The third choice is to build a ".NET" program, or CLI type application, that uses "just in time" compiling and other "nice" features that Microsoft hopes will trap you into using just their products. You can read about all this in Horton Ch1. We won't do that. So, in the book, it's safe to ignore the CLI stuff, which comes later in each chapter.

If you start up by creating a new "clean" project (as described by Horton in Chapter 1, you can select options to use static libraries, whether to use Unicode libraries. I don't know whether not using Unicode libraries helps or not. I doubt the Unicode version of anything is compatible with the good old tried and true stdio functions like printf. Horton is trying to get us to grow up and use Unicode and more 'modern" functions to do stuff. I'm resisting. I just don't have time to explore and learn all that stuff in the middle of all else that needs doing. If you are going to write C++ code seriously, especially for creating Windows applications, you probably need to spin up on all this stuff. But not now.

Some basics:

Horton uses "cout" and "cin" for input and output. These functions are defined for C++ but not C, although some C compilers might accept them. I prefer the old fashioned printf and scanf and their relatives fprintf, sprintf, fscanf, and sscanf. You will get warnings if you use these from the compiler, which is trying to get you to use more modern techniques. Fine, if you can figure them out and get them to work, you are welcome and even encouraged to do so. But remember that when you go back to the microcontroller you are in C, not C++ (although you CAN do C++ for the microcontroller using CodeWarrior. It's just a bit too complex for what we need to do on the microcontrollers. Under Windows we don't really have a choice.

Horton does have some stuff on Unicode in Chapter 2. Notice the special notation needed for Unicode strings.

Variables and basic structures:

Horton isn't clear on whether an int is short (he says so when first introducing ints) or long (which he says is true of the current version, later). Under Code warrior, an int for the HCS08 is short, but I believe you will find that an int for the MCF51JM128 is long. If you are uncertain and it matters, use a short or long, not an int. You do have to use ints for compatibility with many existing functions.

Note that he says that a standard for naming variables is that an initial capital indicates a class, while an initial lower case letter indicates a variable. As I've mentioned, a capital elsewhere in the variable name is a convention often used for global variables, while all lower case is for local variables, what he calls "automatic" variables. We know those as stack variables.

He describes how you can use braces to define a "block" of code within a function and have variables local to just that block and not the whole function. I don't know if this is C or just C++, and how recent. I do think this is done by both the Codewarrior for Microcontrollers C compiler and the Visual C++ compiler. I think it's a bit dangerous. I'd prefer to declare all local variables at the start of a function and give them scope for the whole function, and not have variables local to a block. It can lead to some confusion. (On the other hand, I tend to re-use variables, such as using the same local variable i for the loop counter in several different loops, and CS purists would probably frown on that.) If you are programming for a microcontroller, space and speed efficiency matter. If you are programming a GUI under Windows, they don't. (That's one reason Windows programs are so big; nobody cares about memory or disk use efficiency.)

Horton mentions the "bool" type for C++. Don't bother with it. In C an int or char can be used instead, and if you somewhere in a header have #define TRUE 1 and #define FALSE 0 you can use that instead. I'd also stay away from enumerators and other abstractions that help hide what's actually going on.

Notice the discussion of casts. We will occasionally use casts. You will see that in my stuff on how to use the edit boxes in how to manipulate character data that must be sent to Windows as Unicode. I and most of the code I see uses the "old style" casts. In fact, I don't think I have yet seen a piece of code outside this book, and maybe some other textbooks, that uses the new style. The new style may be "preferred" since it is more explicit, and thus leaves less room for mistakes. But I'm going to use the old style.

In Chapter 3 you will find the "switch" statement (sometimes called a "case" statement). There are times when these are handy, but they are trickier than using if / elseif / else constructs because you have to break off each case runs into the next one. I'd avoid the "switch" unless you have very good reason to use it, and then try to be very careful.

In Chapter 4 (a very important chapter since it deals with pointers and strings and such) you meet multidimensional arrays. These things are tricky, and I don't think we will need them for what we are doing. One of the things that makes them tricky is that they were done differently in the original K&R version of C. If you don't need to get cute with these, such as accessing them also in another manner like as one single array (using a cast), they shouldn't be too difficult to use. Often times an array of strings is stored as an array of pointers to strings instead, so you only need the storage space for each individual string (but you have the extra pointer as overhead). That's the old K&R method. It's described later in Chapter 4. On the microcontroller it matters a lot whether a string (particularly) is a constant or a variable. There's a lot more ROM than RAM available. Under Windows it's not usually important.

Dynamic Memory Allocation is a tricky business. I don't think we need to do it, and I do not plan to cover it in the course. But for larger more involved applications it is often very important, and a source of no small number of bugs. If you do dynamic memory allocation on the microcontroller, I'm not sure where the memory is allocated from. It probably depends on the allocation function (malloc vs alloc in C). In C++ you ask for a "new" object. That's not necessary if the object is statically defined (as our App is). I believe the Dialog is dynamically allocated in the sample program.

The C functions for manipulating strings are found in <string.h>. I use those rather than the C++ library functions; they may be mostly the same. I've seen a lot of them in C. But, be careful about this if you have said to use Unicode; there may be a separate Unicode version of each and you don't want to use Unicode functions with ascii char strings.

You can see just how different programming is if you have to use CLI by comparing the use of strings in that environment to the traditional ANSI C++ earlier in Chapter 4.

Chapter 5 on functions is very important. The Chapter 6 material, where Horton gets into pointers to functions and such, is something we have seen, but less often. You will recall that the arguments passed to the terminal in the HCS08JM60 example upon initialization were three pointers to functions for putchar(), getchar(), and kbhit(). In C++ under Windows there are a good many places where pointers to functions have to be used, but in most cases you will not have to do anything new or creative with these. The exceptions stuff is likewise a peripheral topic that hopefully we won't need. If there is a possibility of something bad happening, like a zero pointer value or division by zero, use an if statement to check and if the error is detected print an error message then exit or do something else appropriate. For an example, look at what the sample application does if HIDOpen returns a zero (NULL). Don't bother with function overloading unless you have very, very good reason. Likewise, function templates is something we shouldn't need to get into.

Classes: This is C++

Most of the stuff in Chapter 1-6 is applicable to both C and C++. The "struct" in C is similar to a "class" in C++. But there is a lot more going on in C++. This chapter is important. You do have to use structs sometimes in C++, particularly the Rect (rectangle) and other basic GUI elements, but in most cases we won't have to go in at that level. (We also shouldn't need to do linked lists or trees or all that other stuff with dynamic objects that programmers often get excited about.) This stuff about classes is important because most of the stuff we interact with under Windows in C++ are things defined in classes. Our application itself is a class, and there is one instance of it, the application we are running. Likewise the dialog is a class with one instance. The check buttons are each instances of a class of object. Each class has its own functions, and each instance has its own variables (normally).

Most of the stuff in Chapter 8 we don't have to dive into too deeply. Unions are interesting. C also has unions, and they are used to make the bits or subfields of a register separately available, or the register as a whole, in the C include files that we use on the microcontroller. But union notation is awkward, so the developers of these files have used defines cleverly so you, the programmer, doesn't have to use or even be aware of the unions. Things like overloading are of importance in C++ but I don't think they are important to what we will be doing. There's a lot of stuff on strings but I believe it is applicable mostly to C++ and not C. Class templates are something that is a factor in programming under Windows but, again, I don't think it's anything we need to spend time on.

One important point in Chapter 8 (p455) is the use if #ifndef / #endif to make sure only one copy of something is found in your header files. You'll see a lot of this in both the microcontroller and Windows code. The "Organize your Code" section is worth reading.

Chapter 9 on inheritance is important to really understanding a lot of the stuff that is going on with inherited objects like windows, your application, various controls under Windows. A lot of the mess in this concerns private vs public accessibility, which can get complicated. For us, we will generally be accessing things using public member functions; I don't thin that will be a problem. We don't have to worry about constructors and destructors. Virtual functions are worth reading about – it's an important capability when you derive one class from another. But I can't think of a case where this is important in our immediate project which works with classes that are already defined. Yeah, this stuff gets pretty complicated. Fortunately what we need to do won't require us to create these things so much as use what's already provided.

Other stuff before we get to Windows:

Chapter 10 on the Standard Template Library: In a C++ course you'd spend a good bit of time on this but I don't think we need any of it.

Read Chapter 11 or at least skim it on debugging techniques. This talks about the use of the Visual Studio 2008 debugger among other things.