if (Language != C) // Testing with Assembly, C++, and other languages

You've surely noticed that this course is entirely based in C. This is so for a handful of reasons. First of all, the primary audience for this course is embedded and systems level developers. C is the obvious language of instruction for our audience. Second, C is the granddaddy of so many programming languages that its syntax is essentially universal to just about any style of instruction on software concepts. Further, because of C's influence on software development in general the concepts we demonstrate should translate reasonably well to a host of other development environments.

What About Testing in Assembly?

Inevitably, this question arises. What about testing in assembly? Simple fact of the matter is that you will not find general purpose frameworks for testing in assembly. You *might* find one-off, purpose-built tooling out there. Maybe. **Apart from specifically addressing unit testing assembly code, the discussion below has equal value simply in presenting the breadth and creativity of thinking test-first.**

Testing in assembly with the concepts we show in this course is difficult to generalize—maybe even impossible. First of all, any given macro assembler is intimately tied to the processor it supports. In effect, then, every platform has its own assembler-level programming language. This does not lend itself to the economies of creating general purpose frameworks. More importantly, programming in assembly is not the more-or-less pure function / method / subroutine / coroutine model of high level languages. That is to say, a block of assembly programming is not nearly as walled off from other blocks of programming as is the case in higher level languages. A five line block of assembly code can just as easily do five unrelated operations as it can accomplish a logically coherent routine. And even that five line coherent routine can unintentionally impact and break other routines.

To thoroughly test assembly with test cases built up of assertions as we demonstrate in this class, you would need to assert the expected state of all registers after each operation. That's nutters, of course.

All this being said, you can still apply the thinking you learn in this course to working with assembly. Your circumstances and tools and requirements will dictate what is and is not practical and valuable. Below are some rough ideas to prime your thinking:

- Get creative with your source code text file management and script a system able to place blocks of assembly code inside C functions (incidentally, C allows this). Run test cases against those blocks of assembly within C functions. This will motivate a careful separation of concerns and thoughtfulness as to the responsibilities and side effects of blocks of assembly code. Your testing strategy here should include mixing the order of calling routines to protect against an assembly routine inadvertently corrupting the operation of another routine.

- Develop a scripting environment for a debugger / simulator / emulator to set up, run, and verify ad-hoc test cases. Again, this will motivate good separation of concerns and thoughtfulness as to how your assembly is structured.
- Develop a means to capture and compare blocks of register values as test cases. In this scenario you would thoroughly test and debug certain routines by hand and then compare the before and after states of registers in a suite of regression tests. This approach has value in flagging unexpected changes to register values due to future development. While not truly test-first development, this technique is sure to guide your thinking on how to craft your routines for testability. Designing for testability tends to lead to better design, period.

All of the above require careful consideration of tradeoffs. In all likelihood you will not be able to unit test all your assembly code in even a moderately sized project. But keep in mind that unit testing is not a magic spell for your projects. Unit testing is simply a smart way to further enable already smart developers trying to work smartly.

What About Testing in C++?

C++ is not as prevalent in embedded and systems level programming as C. Nonetheless, it certainly has a presence in these domains. The high-level concepts we present in this course all apply to C++ equally as well as to C. That said, the hairy beast that is C++ tends to complicate the tooling necessary to accomplish test-first programming.

In theory, C++ should be even better suited to unit testing than is C—particularly with regard to Interaction-based Testing where polymorphism and object instantiation should allow simple creation of mocks. In actuality, C++ is such a complex language and toolchain that it often proves quite difficult to work with in a test-first / automated unit testing capacity.

Developers have created multiple well-supported C++ testing frameworks. Run a simple web search and you will find them quickly. One framework may make clever use of the templating engine in the C++ preprocessor while another framework may rely heavily on runtime use of friend classes. C++ is rich and thus allows a variety of philosophies to be expressed in testing frameworks. Each approach as codified in a framework tends to have at least one complication that will negatively impact your testing workflow. We will not address these complications specifically here. Rather, we wish only to set your expectations as to what you will find.

In summary, yes, you can absolutely use what we teach in this course with C++. However, be aware that you must carefully compare the available frameworks for your circumstances and be prepared to pull your hair out more than you expected going in. Collect opinions from trusted colleagues and/or exercise multiple frameworks with some of your production code to understand their idiosyncrasies.

What About Testing in __________?

You can safely assume that just about any high level programming language has one or more available testing frameworks. A good place to start looking is the list maintained as a Wikipedia article.

In general, the thinking and approach we present in this course apply in all these other programming language contexts as well. That said, there are different philosophies and approaches in codifying unit testing as a framework. Some frameworks focus on strong Behavior-Driven Development (BDD) derived templates and syntax. Others strive to present a testing proto-language abstracted away from the source code language. Of course, since we're talking about software development here, capturing expectations, executing assertions, and generating mocks can all be achieved in more than one way in the same programming language. Popular languages may have inspired the development of numerous frameworks, or the community may have converged on a single de facto standard. Options abound… hunting for the right fit may take a while.

As a general rule, the ease and speed of your unit testing tooling setup and workflow are directly correlated to the dynamic features of the programming language with which you are working. That is, scripting languages and those languages with some sort of runtime / virtual machine (i.e. supporting reflection) tend to be easier to work with in a unit testing capacity than purely compiled languages.

Obviously, embedded and systems level programming may not be ideally suited to scripting languages and byte code running on virtual machines. As in all programming choices, the tradeoff is between performance and flexibility-expressivity-power. The tools we present for testing in C accomplish their advanced features by relying on C's preprocessor and a Ruby application for code parsing, code generation, and build management. You are safe to assume that you can assemble a testing framework for just about any programming context. The work needed to assemble that framework and the hurdles in using it day-to-day decrease with the level of dynamism in the programming language you use.