

Jeffrey A. Vaughan
Teaching Statement
January 5, 2012

I have worked as a teaching assistant for seven semesters, four as an undergraduate and three in grad school. Twice, I have had the pleasure and challenge of designing and teaching my own course, an undergraduate seminar on C# programming. These experiences, as well as own experiences as a student, have shaped my views of teaching and education.

Assignments. Some computer science projects are valuable learning experiences; others produce only sleep-deprived frustration. Understanding how students relate to course work is, I believe, crucial to designing effective assignments and syllabi. As a TA for Cornell's data structures and functional programming course (CS 312), I observed students enthusiastically tackling hard problems: web indexing, Lempel-Ziv decompression, and programming language implementation. These well-designed projects appealed to different individuals' innate curiosity, and compelled students to think deeply about serious computer science. In contrast, I saw my C# class frustrated by a project based on the RSA cryptosystem, where the interesting cryptography content was obscured by data conversion technicalities. More successfully, I asked the class to implement the perceptron algorithm and use it to classify handwritten digits as either "3" or "5." Students engaged with this task on several levels: applications-minded students enjoyed the practical problem, the mathematically inclined experimented with a beautiful algorithm, and hackers learned about relevant persistent data structures. These experiences helped me to understand how carefully-considered assignments can motivate and inspire students, and I am eager to apply this to my own teaching.

Classroom. I am accustomed to delivering and listening to PowerPoint talks and initially lectured using slides in my C# class. However, in my small classroom, the format hindered my ability to connect with students. As an experiment, I transitioned to a lights-on classroom. I led class using a mix of the chalkboard (great for working examples and taking discussion notes), a Visual Studio buffer (demos and live tests of student suggestions) and, yes, slides (definitions and diagrams). Success! The students were more engaged, and I more relaxed, after the switch.

Of course, I am continuing to refine my classroom style. For instance, I'm currently looking for a good way to encourage class participation. I find that a handful of students will volunteer to answer almost every question I ask—but I already know those students are keeping up. In contrast, cold calling can induce a discussion-killing, deer-in-the-headlights panic. I recently attended a seminar where the speaker asked each student a question in turn, eventually traversing the entire room. This got everyone speaking, gave students time to contemplate their answers, and avoided cold-call panic. I adopted this strategy in my course and found it to be qualified success. Detailed technical questions—"What's the next line in this function?"—worked naturally, but the format proved too restrictive for some higher-level topics—"What does it mean for a program to be correct?" This experience underscored the close connection between a classroom discussion's subject and its form, a lesson I'm still learning how to exploit.

Broad engagement. Introductory computer science classes are, in my experience, primarily about programming. I worry that this risks driving smart underclassmen out of computer science (perhaps into mathematics, economics, or electrical engineering) and perpetuates stereotypes about the field. In contrast, I

believe that we should introduce computer *science*—the fundamental concepts of algorithms, theory, architecture, etc.—much earlier. While I am hopeful that full-scale curriculum revisions will eventually realize this completely, such changes are too big, too important, and too contentious to happen overnight. For now, I strive to integrate “big ideas” into my teaching whenever possible. For instance, when teaching C#, I found that machine learning algorithms inform the discussion of immutable classes in object-oriented environments. And even the simple distinction between `foreach` loops (which terminate when iterating over finite structures) and `while` loops (which can trivially diverge) connects to deep concepts like the halting problem and Rice’s theorem.

Teaching interests. I’m particularly interested in teaching security, systems, programming languages, and compilers at the undergraduate level, and security and programming languages at the graduate level.