

Designing a first-year curriculum for CSE

John Plaice

May 20, 2005

Abstract

Learning to program and to produce robust, working software is a difficult task and requires much time and energy. The process can be substantially improved by taking a systematic approach to software development, ensuring a comprehensive skillset is given to the student, enabling the potential for real progress in subsequent years.

1 Introduction

In this preliminary essay, I propose an approach to first-year teaching that will ensure that most students end up as competent programmers and software developers. Instead of focusing on grand ideas such as *design*, I believe that we should take a systematic, *bottom-up* approach to teaching first- and second-year students that ensures that they can build software reliably for the higher-year courses. Design is something you learn at the *end* of a computer science degree, or at least in second year. There is a key difference between a project that is well designed and one that is *well organised*.

I am writing this proposal before knowing whether there will be a common first year for all engineering programs or not. However, it seems to me that that issue is orthogonal to the issue of what should be taught. I do not believe that we should be teaching software engineers, computer engineers or electrical engineers differently about basic programming techniques, the need for software repositories, or how to prepare software builds and test suites. In the end, there really are only two kinds of software: software that *works* and software that *doesn't*.¹

Before I get into detailed proposals, I should note what I consider to be key aspects of software, which are often ignored, glossed over or quite simply denied.

First, a piece of software is not virtual, it is *concrete*. Just as a manufactured machine is made up of various plastic and metal components, each with its own utility, so software is also made up of components, called *files* on most current operating systems. And, just as with machines, where you can get the exact part number and description for every component of every version of a particular model, so with software we should be able to completely track every file used to produce a piece of software. Therefore, students need to be introduced, right from the very beginning, to the use of software repositories.

Second, a piece of software will include many different kinds of file, each used for different purposes. Some are source files, some are build files, some are data files, some are documentation files, etc. Each different kind of file has its own structure. For example, source files include, among other things, definitions for classes, functions and variables. Therefore, students need to be introduced to software configuration and the complexities of automatic builds.

Third, software is to be run on a *machine* called a computer. No matter how powerful, how fast or how large the computer, it is still just a very fast mechanical device, where everything is ultimately encoded and stored as sequences of 0's and 1's. Students must learn how to use the C "plain old data types", as well as to build more complex data structures, and how these are to be serialized when stored in files. They must learn how to manage memory, pointers, and their generalizations, iterators. In other words,, students must to learn to program.

¹There are 10 kinds of people. Those who understand binary and those who don't. ANON.

Fourth, it is a common perception that it is difficult to get software right, no matter what the programming environment. We need to endow students with a level of developer's confidence that dispels this mindset, prior to second year. In particular, it is important for them to develop validation techniques for everything that they write. Students need to learn, right from the very beginning, to develop test suites and other validation mechanisms for their programs.

Finally, successful software lasts a long time. It will be modified over time by people who have never met the original developers. Therefore, students must learn to write *readable* software.

The above points might appear to be obvious, but they are important. For example, I have met a number of students in third year who cannot imagine a program that is going to read data files in a directory different from the one in which the binary is residing. You can be sure that if the students are not taught these basic concepts, many of them will *never* pick them up properly.

If the above topics are mastered, then students can move off in many different directions. They can learn to use integrated development environments, automatic code generators, to write sophisticated algorithms, to learn about hardware-software codesign, what have you. But they must master the above techniques before they can move on, because if they don't then we will have to re-teach these methods, piecemeal in third and fourth year, and they still won't learn them until they are forced to when they get to industry or become our Ph.D. students.

2 What first-year should not be

I do not believe that the point of the first-year curriculum, at least on the beginning, is to focus on requirements engineering, design, refinement or abstraction, all of which are top-down concepts. One cannot elucidate requirements if one does not know what a while-loop is. One cannot work at an abstract level if one does not understand basic concrete concepts. We must begin bottom-up and make sure that people get things right. If this is done properly, then we can move on to these higher-level concepts.

3 Two fundamental concepts

A huge proportion of today's computing takes place using two fundamental concepts. These are the von Neumann architecture and the Unix file-process model of computing. The first refers to the idea of a computer composed of an I/O unit, a memory and a processor, where programs and data are loaded into the same memory. The second refers to the idea that data is passive, residing in files, and that processes read data from files, perform processing and then write new data out to files. All first-year students should learn and assimilate these two fundamental concepts, and learn to write programs assuming that they are fundamental.

When we move on to concurrent computing, there are of course variations of these models or new models altogether: client-server, multi-threading, symmetric multiprocessing, middleware systems, and so on. These can be acquired in second year or later.

4 Choosing the right programming language

In the debate about programming language that took place towards the end of last year, there were broadly speaking two camps. Those in favour of retaining Haskell as the first-year programming language, and those opposed. Those in favour claimed over and over that programming language did not matter, just so long as it was Haskell. Of those who opposed using Haskell as first-year language, the three main proposed languages were C, C++ and Java.

In my opinion, all of these three languages should be learned. However, C++ should be understood as the reference language among the three. The reasons are clear:

- Apart from the semantics of the `enum` construct, C++ is a superset of C, so it is possible to write low-level C code and have it run correctly through the C++ compiler.

- The C++ type checker is much more powerful than the C type checker. The C++ linker does type checking of function calls across separate compilation units, making it much safer.
- The C++ language has exceptions, which C does not.
- The C++ language includes templates, which correspond to parametric types in the semantic world. However, the C++ templates are much more powerful than the type system available in any other readily available language, because templates can take as arguments not just types, but also constant values, thereby allowing for very sophisticated specialization of generic code at compile-time. You can have your cake and eat it too: high-level algorithms with arbitrarily precise specializations in order to improve performance.
- The C++ Standard Template Library includes a number of useful container classes, allowing programmers to not use fixed-size arrays, the most common source of bugs in code written by students in my classes.
- The Boost group (<http://www.boost.org>) provides many free peer-reviewed portable extensions to the Standard Template Library, including unit test frameworks, grammar generators, portable multi-threading, metaprogramming (compile-time programming with templates), and so on.
- Both C and C++ get floating point right, properly implementing IEEE 754. This cannot be said for Java.
- If one has learned to program in C++, one can easily move on to Java, or any of several scripting languages, such as Perl, Python or Ruby.
- C++ is an ISO standard. It was created at AT&T Bell Labs, but it does not belong to any company, as do Java (Sun) or C# (Microsoft).

5 The first problem

The first problem that students should be expected to work on should be something that is not technically difficult, but for which the supporting infrastructure is non-trivial. As a result, the students can learn how to manage all of the things that go with producing software, and they can develop the self-discipline that is necessary to become real engineers.

The first class should introduce a version control system, such as `svn`, which works well both locally, through the host file system, and remotely, using `ssh` or an Apache Web server. All tutorials, labs, assignments, specifications, etc., can be passed through the version control system. Even submissions could be done using the version control system. How far an academic wishes to go in this direction is unclear. However, students should be expected to use a version control system to maintain records of their work.

Once students know how to use their version control system, then they can move on to programming. We take a simple problem, like a four-function calculator. The algorithmics are not difficult, so the students can spend a lot of time on everything around the algorithms. How to store information in files, how to read information from the command line and from files, how to process it, do error and bounds checking on input, doing the calculations, handling anomalous situations, outputting information back into files.

In addition to the core program, the software needs to incorporate the necessary tools for building it. At first, a simple `Makefile`, then on to a full `configure-make` infrastructure automatically generated by `autoconf-automake`.

Furthermore, the students need to build a testing framework for their program. This includes unit-testing, using, say, the Boost `test` infrastructure, as well as black-box testing, managed using a setup such as the `autotest` that comes with the `autoconf` setup.

So, after half a term or so, the students know that when they type `give` on their automatically generated tar ball, that what they are submitting will work right out the box.

6 Learning the bits and bytes

The remainder of the first term should be present all of the basic atoms of the current computing world. Here are some key ones:

- twos complement integers using differing number of bits;
- IEEE-754 floating-point arithmetic;
- colours: RGB for light, YCMK for print;
- character sets: ASCII (7 bits), ISO-8859-1 (Latin-1, 8 bits), ISO-10646 (Unicode, 21 bits);
- binary, octal and hexadecimal notation.

The above examples also offer a great opportunity to introduce the debugger, which needs to be referred to in all subsequent courses.

7 Moving on

The second term needs to focus on higher-level data structures, at two levels. First is the use of standard containers and algorithms, as provided by the C++ STL. Second is the design and implementation of simple data structures and memory management, and *error management*, through the proper use of exceptions.

At the end of the first year, the students will know enough about the low-level aspects of ordinary software development that they can take a course in hardware and intuit the relationship between hardware and software. The students will also be capable of putting together proper code so that if they are asked to write more difficult algorithms in second year, then they won't have to worry about whether or not they can manage the software part: they will be able to focus on the algorithms.

8 Conclusion

This little piece needs to be elaborated upon. However, I believe that it shows that we can design a quality curriculum around a procedural language right upon entry from high school, that ensures that the vast majority of the students learn to program properly. It will make the teaching of higher-year courses much more interesting.