Programming in an undergraduate CS curriculum

Bjarne Stroustrup
Texas A&M University
bs@cs.tamu.edu
www.research.att.com/~bs

Abstract

This note argues for a fairly classical undergraduate computer science (CS) curriculum where "software" (programming and related topics) takes a bigger role than is often the case. The discussion is based partly on experience with an undergraduate curriculum change at Texas A&M University and with developing a new freshman programming course. That freshman course is the central topic of this note. Based on industrial experience, it is argued that the primary aim of a university education in the area of "software" is to be a foundation for professional work. The primary design criterion for the freshman (first year) programming course is to make it a good start at that. Caveat: the opinions expressed about the needed improvements of and directions for software education is based on personal experience rather than hard data.

1 Introduction: Problems

My perspective is that of an industrial researcher and manager (24 years at Bell Labs; 7 of those as a department head) who has now spent 6 years in academia. I see a mismatch between what universities produce and what industry needs (not just what industry says it needs). When I say "industry" I obviously simplify, but I base my simplification on talks with dozens of representatives of (primarily, but not exclusively, US) industry a year over the last 30 years or so.

Industry wants computer science and computer engineering graduates to build systems consisting largely of software (at least initially in their careers). Many graduates have essentially no education or training in that outside their hobbyist activities. In particular, most see programming as a minimal effort to complete homework and rarely take a longer-term view involving use of their code by others and maintenance. Also, many students fail to connect what they learn in one class to what they learn in another. That way, you can (and often do) see students with high grades in algorithms, data structures, and software engineering hack solutions in an operating systems course with total disregard for data structures and algorithms, resulting in a poorly performing unmaintainable mess.

For many, "programming" has become a strange combination of unprincipled hacking and invoking other people's libraries (with only the vaguest idea of what's going on). The notions of "maintenance" and "code quality" are at best purely academic. Consequently, many in industry despair over the difficulty of finding graduates who understand "systems" and "can architect software."

This is not all or even primarily the fault of the students. The academic curriculum is crowded with topics of undisputed importance and in the resulting competition for time, practical issues and the development of time-consuming practical skills (such as design for testing) lose out to more classical academic subjects (such as mathematical analysis of algorithms) or fashionable research topics (such as subsurface luminosity). However, to remain an applied discipline – as it has been from its inception – computer science must emphasize software development and CS programs must allocate time for student skills to mature. If we don't, we are like a music department that does not require musicians to practice before a concert or an athletics department that "trains" its athletes primarily through lectures.

For the good of both industry and academia, we must do better.

2 What do we want?

A university education should lead to a well-rounded personality, provide a solid grounding in an academic field, and be a reasonable preparation for a job (or grad school entry). I consider a basic knowledge of history, art, math, and science (e.g., biology and physics) essential, so I'm not going to argue for more CS at the expense of a liberal education. I just wish the liberal arts curricula similarly went out of their way to avoid producing scientific ignoramuses.

The majority of CS graduates do not become professors, so I consider preparation for a career in industry crucial. I use the term "industry" in its broadest sense, including non-profit organizations and government. Industry does not want "scientists" in large numbers (I have heard repeated reports of recruiters denying interviews to "computer scientists" for

WCCCE '09 May 1-2, 2009, Vancouver, British Columbia, Canada.

that flawed reason), engineers maybe (for some definition of "engineer"), but they definitely want "developers." Consequently, I consider producing software professionals (for some definition of "professional") the primary aim. Hardware is of course also essential and should be part of every curriculum, but in this context, hardware is "someone else's problem" so I won't consider it further.

I (emphatically) do not suggest that universities should simply produce what the majority of industry recruiters ask for: developers trained in the latest fashionable languages, tools, and methodologies. That would do harm to both academia and industry. Fashions come and go so rapidly that only a solid grasp of the fundamentals of CS and software development have lasting value. Training developers – rather than educating computer scientists (under whichever specific label you prefer) – would lead to a stream of employees being disposed of as fashions changed. That is neither moral nor cost effective. The alternative is a focus on fundamentals combined with the development of practical skills based on them.

The days where you could learn all that you needed for a career during four years in a university are long gone. A university education is just one stage (although an important stage) in a life-long education. Thus, there has to be an emphasis on providing and reinforcing dedication to and skills for self-education.

3 Computer science

Programming obviously isn't all there is to CS, however popular that misconception is among many people in the wider community. In fact, few tasks in software can be done well without a reasonable knowledge of algorithms, data structures, and machine architecture. For many tasks, we must add operating systems, networking, human-computer interfaces, graphics, and/or security to the requirements. And what about theory, compilers, math, etc.?

I find that CS professors often overreact to the inaccurate popular image of the software developer as a lonely guy with "no life" hacking code all night. To counter, they cry "Computer Science is *not* programming!" That's true of course, but that reaction can lead to a serious weakening of programming skills as some adopt the snobbish attitude "we don't teach programming; we teach computer science" and leave practical software development skills untaught.

Programming is the primary means of making ideas into reality using computers. Everything that runs on a computer is directly or indirectly expressed in a programming language. Most people don't have to program; their needs are well served by pointing, clicking, dragging, writing HTML (usually using tools), etc. However, to have a solid understanding of computer systems you need a practical and theoretical understanding of programming, programming languages, and other tools supporting the development of trustworthy code. Preferably, this understanding extends to several kinds of languages (e.g., declarative, scripting, general purpose, and close to the hardware) and several kinds of applications (e.g., embedded systems, text manipulation, small commercial application, scientific computation); language bigots do not make good professionals.

4 Software and the software curriculum

I see software as an artifact that is interesting in its own right (and not just for what it does) with a structure that can aid or hinder its growth and usability. I'm not alone in that view so parts the new Texas A&M University CS curriculum reflects it.

The study of "software" includes

- software engineering on a small group scale. The design, implementation, and maintenance of million-line systems are beyond our "software" curriculum and left for people who care to specialize in that and especially for industry to handle. A balanced 4-year undergraduate program can do little on this scale; though it must try to prepare students for scaling up.
- the use of programming languages (note the plural). However, most language implementation, language theory, and comparative languages studies are left for more specialized courses. For some of us that is a great loss, but something has to give.
- individual and group projects done at each level starting in the first programming course and repeated with increasing difficulty in every software course. These projects are central to teaching the beginnings of the "higher level" project management and software engineering skills. This is where tools such as test frameworks and source control systems find their natural homes.

The fundamental notion is that programming has a theoretical basis but is also a craft, like playing the violin. We need to teach both. We have seen too many examples of unnecessarily ghastly code – expensively and laboriously produced by bright, well-educated individuals who just never were given guidance in how to structure a program for further development and maintenance. Conversely, talented, mostly self-taught, programmers struggle for lack of a theoretical foundation for their work.

This leads to the question of what to teach first: the theoretical basis or the craft? Since there is no way a student can appreciate the problems of writing correct software or maintaining large code bases without having programming experience, we have to start by writing code. Only through trying to write code and debugging it do people get to appreciate the magnitude of difficulty in producing correct software. Furthermore, only by facing the problems of evolving an existing code base do people appreciate the value of clean design, supporting tools, and testing. Long lectures on software engineering to people with a weak software background are at best ineffective and at worst instill a dislike for programming as a low-level activity unworthy of serious attention ("a mere implementation detail") and/or of software engineering as "irrelevant and abstract." So we initially approach programming as a craft with an emphasis on how to get real-world code sufficiently good for people's lives to depend on it. That implies a constant attention to requirements of correctness and practical techniques of how to meet them. That way, the more abstract principles emerge naturally from concrete needs.

Our "software curriculum" is a sequence of courses:

- Introductory programming (using C++[2])
- Data structures and algorithms
- Programming languages (at least two that are not C++)
- Design studio (using at least two languages out of the three taught)

The "design studio" is primarily project based and aimed at pulling together what is taught in the other courses. Most topics are taught repeatedly in increasing levels of detail and rigor. These four courses are ideally fitted into the first two years of study and are designed to provide a student the basic knowledge and expertise to qualify for and benefit from an industrial internship.

This is of course just part of a broader curriculum offering machine architecture, discrete mathematics, human-machine interfaces, compilers, and more advanced and specialized courses. The software program is often completed through a project-based "capstone design course" taken in the final year.

5 The first programming language

The choice of a first language is always controversial. TAMU used Java for a few years and the experiences were mixed and the comments from industry interviewing and/or hiring students discouraging. This fitted a pattern I had seen in many places in industry: a desire for greater knowledge of "systems" and "machines" combined with an emphasis on performance that didn't match what the students had been led to expect. A further constraint was that the Electrical Engineering department and the Computer Engineering faculty insisted on "not a teaching language, but something close to the hardware, preferably C/C++."

It is often pointed out that the choice of programming language is less important that the choices of programming philosophy, design methodology, and teaching approach. Changing programming languages is at best part of a solution. However, a programming language carries with it a whole host of assumptions, attitudes, and an emphasis on select application areas. Part of the task of teaching programming is to make such cultural factors explicit and to use the initial language to its best effect in areas where it is suitable. Teachers must try to take the focus away from programming language technicalities and focus on more general issues, such as software development. However, a first language must be chosen and used well.

Given my background, our choice of C++ is unsurprising, but we try to avoid language bigotry by insisting that all CS graduates learn at least three languages. C++ has the strengths of being very widely used, having an ISO standard[1], being well supported on all platforms (including the embedded systems platforms), and supporting the major programming styles (paradigms). Its obvious weaknesses are complexity, archaic features, and the dangers of accidentally violating the type system. We decided to compensate for the weaknesses by an emphasis on type safety, encapsulation of "dangerous features," and use of libraries. Every line of code presented (except examples of what never to do, but including GUI) runs on all major platforms. Every time we have given the freshman course the students have used a mix of Windows, Macs, and Linux systems.

A language, its compiler, and basic development platform is only the first and most basic tools for a software developer. The first programming course does not progress beyond this (unless you count downloading, installing, and using a software library that is not part of the standard). My view is that using a general-purpose programming language (rather than a simpler "teaching language") and common industrial programming environments (rather than specialized educational environments) give students as much initial exposure to tools as they can handle. Add more tools, and many will get distracted from the code itself and develop an unhealthy dependency on ill-understood, non-standard, and often proprietary facilities.

6 The introductory programming class

My basic idea for the design of the freshman programming class was to work backwards from what is required to start a first project aimed for use by others. That list of requirements defines the ideal set of topics to be covered. Naturally, we can't completely cover all that (even assuming suitable supervision for that hypothetical next project). You couldn't train a plumber in three months let alone an acceptable high-school violinist. To compare, learning the basics of a natural language takes upwards of three years. Yet, we succeed in assembling a toolset of concepts and techniques. Students have reported that they have put what they learned to good use on their first real projects.

The freshman programming class is based on a book that I developed concurrently with the course and refined based on experience with the course. The book is entitled *Programming – Principles and Practice using C++*[3]. Its preface, table of contents, lecture slides, and other supporting materials can be found here: `http://www.stroustrup.com/Programming.html`.

Our approach is "depth first" in the sense that it quickly moves through a series of basic techniques, concepts, and language supports before broadening out for a more complete understanding. The first 11 chapters/lectures (about 6 weeks) cover objects, types and values, computation, debugging, error handling, the development of a "significant program" (a desk calculator) and its completion through redesign, extension of functionality, and testing. Language-technical aspects include the design of functions and classes. Finally, interactive and file I/O are explained in some detail. The data types used are bool, char, int, double (a double-precision floating point number type), string (a variable length sequence of characters), and vector (an extensible container of elements). That's "the basics." At this point, the students can (in principle) do simple computations on streams of numbers and/or strings – they are by now dazed and need a break.

There is a constant emphasis on the structure of code (invariants, interface design, error handling, the need to reason about code to ensure correctness, etc.). This is hard on the students, but seems to bear fruit and stop them from obsessing over language details. Concepts and techniques are presented through concrete examples followed by the articulation of an underlying general principle. Typically, the stu-

dents have a hard time grasping the importance of the principles, which are seen as "abstract," so repeated application to concrete examples is essential. The style of the concrete examples reflects the principles and can – when imitated by the students – lead to later understanding.

At the end of this part of the course, the students should have no problems with simple exercises like this:

```
// produce the sum of the integers in "data.txt"
ifstream is("data.txt");
if (!is) error("data file missing");
int sum = 0;
int count = 0;
int x;
while(is>>x) {        // read into x
        sum+=x;
        ++count;
}
cout << "the sum of " << count
        << " elements is " << sum <<"\n";
```

They should also (often somewhat hesitantly) begin to define simple types (classes and enumerations) to simplify their code.

That "break" after "the basics" takes the form of 5 chapters / lectures (about 3 weeks) on graphics and GUI. This actually does refresh the students, increases their level of interest, and makes them work harder. It's mostly graphics (as opposed to GUI) and the students do not perceive it as difficult. Class hierarchies and virtual functions are introduced. That is, the fundamentals of object-oriented programming are presented as a simple response to an obvious need. This is a technique we use repeatedly. After all, much of a first programming course is simply to supply a strange formal notation for what the students learned in primary school, or even before that ("if the light is green, cross the road; otherwise, wait" is not rocket science). By relying on such student knowledge where we can, we gain time needed to deal with concepts and techniques that are genuinely novel to the students. For example, we don't spend much time on simple control structure, arithmetic, or (really) basic geometry. The student *do* know that. The time we gain can be spent on, say, input validation, error handling, and the value of user-defined types (classes and enumerations).

A typical simple exercise would be to define and use a simple graphical class:

```
// define a shape class "Arrow":
class Arrow : public Shape {
public:
   Arrow(Point p1, Point p2)
        // construct an arrow from p1 to p2
        // with default head and tail
   {
        check(p1,p2);  // long enough?
        add(p1);       // store tail point
        add(p2);       // store head point
   }

   int check(int len)
   {
        if (length(p1,p2) < min_lgt)
           error("arrow length too small");
        return len;
   }

   void draw_lines() const;

   int length() const;
   Point point() const; // head point
   void set_point(Point x);

   // ...
private:
   const int min_lgt = 10; // minimal length
   Head hd, tl;            // head and tail
};

// make a few shapes:
Point center(0,0);
Arrow a0(center,Point(20,50));
Vector_ref<Shape> vs;  // container of Shapes

vs.push_back(a0);
vs.push_back(new Arrow(Point(20,50),Point(50,50)));
vs.push_back(new Rectangle(oo,Point(50,50)));
```

Larger examples includes the animation of consecutive approximations of an exponential function and graphing data sets from a file.

The third part of the course (about another 3 weeks) has two sections:

- The first is a detailed explanation of how the C++ standard-library vector is implemented and the second an introduction to containers and algorithms using the STL (including that vector). The STL is the ISO standard library facilities providing containers and supporting computations (algorithms) on sequences of data. The sequences of data can be either conventional input / output or containers of elements (such as the elements of a vector). Explaining vector involves the introduction of pointers, arrays, and C-style manipulation of memory. We did not consider it responsible to leave that out. This section has been expanded and refined over time based on requests from "consumers" of our students.
- The second section introduces the basics of generic programming. The STL provides a contrast to the low-level pointer and array manipulation from the first section with a much higher level approach to algorithms on data structures. We use vectors, lists, sets, and maps with algorithms such as find, sort, and accumulate. Most modern programming languages support roughly equivalent facilities, even if they are typically either built-in (rather than provided in a library) or less general. I consider a working understanding of such basic data structures and algorithms in their colloquial form for a chosen language essential. There have been requests for expansion of this section also, but I don't see how that could be done without cutting something else.

A typical simple exercise at this stage would involve the use of STL containers and algorithms:

```
// print unique words from iname to oname in order
istream is(iname);    // input from iname
ostream os(ofname);   // output to oname
if (!is || !os) error("couldn't open file");

istream_iterator ii(is); // start of input
istream_iterator eos;    // end of input
```

```
ostream_iterator oo(os,"\n"); // start of output
                              // (newline separated)

set<string> b(ii,eos);      // read words from input
                            // into a set
copy(b.begin(),b.end(),oo); // copy words to output
                            // ordered by the set
```

The combination of the STL for storage and algorithms with graphics is a good base for exercises and projects. However, the students have also tried conventional C-style techniques:

```
char* cat(const char* p, char c, const char* q)
   // concatenate p and q separated by c
{
   int lp = strlen(p);
   int lq = strlen(q);
   char* r = new char[lp+lq+2];
   strcpy(r,p);
   r[lp] = c;
   strcpy(r+lp+1,q);
   r[lp+lq+1] = 0;
   return r;
}

char* p = cat("someone",'@',"somewhere");
cout << p;        // "someone@somewhere"
delete[] p;
```

The variety of programming styles can be confusing, but it is minor compared to what is found in real-world code. To help the students cope, we frequently refer back to fundamental principles, to commonly useful styles of code (reflecting those principles), and try to offer guidance about preferences. Style matters.

Usually, there is time for just one more lecture: a presentation of ideals for software followed by a quick trip through the history of programming languages giving examples of how languages have increasingly supported those ideals. This lecture complements an introductory lecture on applications of software and the importance of software and its developers to society. This kind of motivational material is essential as the students are pretty clueless about the role of software in the world. We try to complement it with a sprinkling of "news flashes" in the individual lectures.

This material is very extensive for a first course and the pace quite high, but most students succeed. In addition to the final 3-person 3-week project (running in parallel with lectures) which all students do, we have noticed many students experimenting with private (not homework) projects such as a "catalog" of friends with contact information, comments, and photos. It is not uncommon to find elements in the final projects that were not taught, thus demonstrating student interest in and ability to go further. It would have been nice to give the students sufficient tools for a web interface for such projects, but that's beyond us for now.

There is a fourth part to the book aimed partly as support for projects, partly to support people learning on their own, and partly for extended versions of the course: text manipulation (including standard-library regular expressions), numerics (mostly an N-dimensional matrix class to save people from the horrors of C-style multidimensional arrays), embedded systems programming (mostly low-level memory manipulation and bitfiddling), testing, and a survey of C (there is a *lot* of C code "out there"). Sometimes, we manage an extra lecture or two based on these chapters. That depends on how a class progressed, how many review sessions had to be included, etc.

7 Problems: Execution

What happens when these ideals, ideas, and plans meet a class of 240 freshmen of mixed abilities, aims, and backgrounds? My experience is based on a mixture of EE, CE, and CS students, 60% of whom have programmed before (mostly in high school "advanced placement" CS courses) and 40% have never seen a line of code. TAMU is a good public university, but not an elite institution with the ability to reject most applicants. This course has now been given nine times by a variety of professors to about 1500 students. We are reasonably convinced that the approach scales. A College of Engineering survey showed that the students on this course worked 25% longer hours than average for our engineering school freshman classes, yet reported 20% higher satisfaction. However, we can't provide meaningful quantitative measures of success.

The most common complaint about the course has been that the order of topics is confusing and illogical. This primarily comes from students who have programmed before and have a firm idea of what should be taught and in which order. Typically, "bottom up" or "all C language features first" is seen as "natural" whereas the ordering based on programming needs and principles rather than language features is seen as "wrong and unnatural." To contrast, students who have never programmed before do not have a problem with our early use of standard library facilities (such as iostreams, string, and vector) and do not find the early absence of pointers and arrays strange. Appendices presenting C++ and its standard library in conventional manual order aim to address these comments.

The second most common complaint is that the repeated statements of principles (to achieve correctness, maintainability, etc.) is "over our heads" and "irrelevant for programmers." The latter comment proves the need for an emphasis on professionalism. We address this problem primarily through a close tie between concrete examples (code) and statement of principles. In particular, we (also) present examples of errors to teach the students to recognize both "silly errors" and violations of principles. The students do seem to make fewer "stupid errors."

Textbooks: Before starting to design the freshman programming course, I surveyed a couple of dozen introductory C++ / programming textbooks and saw some patterns I found disturbing. Most could fairly be criticized for teaching C++ more than programming and they tended to do the student a disservice by presenting a very complete and detailed bottom-up view. For example, some present all variants of C++ built-in data types and control structures before presenting meaningful examples of their use. Others present all features present in C before "the extensions" provided by C++ and avoid C++ standard library facilities. In the hands of an uninspired (or weakly prepared) teacher, this bores a good student to tears, presents programming as an unending sequence of obscure technical details, gives a view of C++ an unnecessarily complicated variant of C, and presents cru-

cial higher-level concepts essential for industrial use (e.g., the STL, ways of defining classes, and realistic uses of inheritance) late labeled "advanced" (and consequently typically avoided). In a one-semester course based on such books, students never reach useful programming techniques and are never faced with meaningful challenges. My response was to teach based on notes and turn those notes into a book, which is now the basis of the course as described above. Other software courses face similar challenges. Too many textbooks are either dumbed down, dissociated from real-world practice, or simple how-to-guides that don't expose students to principles.

Students: Mostly, the students are not a problem. Exceptions include students who have programmed before and insist on showing off and to teach other students "cleverer, more efficient, ways" of programming than what is taught, such as

```
char buf[128];
gets(buf);   // Nifty! efficient! simple!
```

rather than

```
string buf;
getline(cin,buf);  // professor's boring stuff
```

(Quick test: Why is that `gets(buf)` a disaster waiting to happen?[1]) If not carefully instructed, TAs (Teaching Assistants) can add to this problem by regurgitating what they have absorbed over the years rather than following the rules for the course.

Having not taught freshmen before, my greatest surprise was that a major component of the course became teaching the students the need to work and to give guidance on how to do that. Many had breezed through high school and expected to do well reading notes or listening to the lectures (not both!) and going to "the labs." The idea of having to concentrate during the lectures and then spend 10 hours a week outside class re-reading the notes and working through exercises was alien.

When we went from using notes to using the finished textbook and also made the complete set of lecture slides available on the web (primarily for the benefit of readers who are not TAMU students), we experienced a most unfortunate drop in attendance. Apparently, many students think that listening to a lecture as well as reading a chapter is a waste of time. I see the redundancy as necessary reinforcement, so I expect this drop in attendance to offset much of the gain from the better material. Since almost all students miss important points when doing a single pass over new material, they will pay for their "saved effort" with longer debug sessions.

Projects: The final 3-person 3-week project (alongside lectures) is essential. That's where everything comes together and where the students get their first taste of project management. It's hard to come up with a sufficiently long non-repeating series of such projects. Examples so far include scrabble, sudoku, and whack-a-mole. Games make good projects because they are perceived as interesting and are open-ended (e.g., improve the GUI interface, add "cleverness" to the computer "player," etc.).

[1]The input may overflow the fixed-size buffer; this used to be the single most common security hole in C code.

Exercises: Most exercises are of the "write a program" variety. So far, we have always been short of good exercises. This problem is decreasing over time, but finding exercises that are challenging (but not too challenging) for the students and also present problems that the students can relate to is hard. In particular, we need more exercises of the sort where the teacher provides a larger program and the student fixes bugs and/or extends functionality. Many exercises involve improving solutions from previous chapters, thus hopefully reinforcing the lesson that code is an artifact that needs a structure to ease modification (a.k.a. "maintenance").

Grading and testing: Obviously, grading of homework of the "write a program" variety is time consuming and tedious. It is also unavoidable and essential to provide students feedback on their coding style. In this, CS has much to learn from the humanities. Inadequate feedback caused by lack of time and attention can undermine the key message of "correctness and systematic testing of code." The fact that we – due to lack of funds and personnel for alternatives – regularly have to use multiple-choice testing is a major problem. Many students will study for the test, and multiple-choice tests are ill suited for testing higher-level skills. Imagine basing a large part of a pianist's or a soccer player's ranking on a multiple-choice test! I consider this analogy between programmers, artists, and athletes fair. My ideal world has much feedback from teachers and no multiple-choice tests. But in the real world, the student/teacher ratio often limits what can be done.

The exact details of grading have varied over time, but this should give an idea.

- 30% Homework and drills; a drill is a simple multi-step programming exercise to ensure that a student has mastered the basic practical aspects of a lecture.
- 45% Three (closed book) multiple-choice exams (Yuck!)
- 20% Final (group) project
- 5% Attendance and class participation (verified by pop quizzes).

We plan to increase the weight given to pop quizzes (to increase attendance) at the expense of the – quite likely counterproductive – exams. This system of assessment is probably the weakest aspect of the course, but we are heavily constrained by local culture.

Teaching assistants: Typically, TAs are grad students that arrive back on campus on the same day as the undergraduates. This makes it really hard to be ready for the "Hello, World!" program on day 1. Departmental machines have to be ready and the TAs gathered together to be minimally trained. It is hard not to stumble and immediately get the homework out of sync with the lectures. Though fundamentally not the TAs' fault, this has been a major problem.

Another problem is that not all TAs attend a lecture and also carefully read the chapter before seeing the first student. Some try to "wing it" by relying on earlier knowledge of C/Java/C++/programming. However, what we teach is not "your father's C++" and our approach to software development is less tolerant of "messy code" than they have been used to, so sometimes they end up sending the message that less rigorous approaches to programming are acceptable.

Professors: For scaling a new approach, we are the weak link. Any mistake by the professor gets amplified by the TAs and the students. In particular, falling back to a slower pace, explaining more of the basics early on, and giving the students a break on exercises are obvious dangers. Students live up to expectations, but are also quite willing to be convinced that a course is "a waste of time" and "too difficult." By slowing down, a professor can significantly delay the point in time where students gain satisfaction from completing a program that actually does something interesting. By slowing down, a professor can seriously delay the point where a motivated student feel the satisfaction of mastery of new knowledge or skill. By slowing down, a professor can trap motivated students into a pattern of inattention, too little work, and alternative activities (not leading to higher-level programming skills). Whatever we do, we should take care not to demotivate the students most likely to become the best software developers and computer scientists. Rather than slowing down, we should – and do – add TA support and review sessions (really catch-up session) for the tail end of the students.

Software: We allow students to use any computer with a reasonably up-to-date C++ compiler. That has worked pretty well. The major problem is that C++ does not have an ISO standard GUI. Instead I had to choose from among the couple of dozen available C++ GUI libraries and toolkits. I chose to use FLTK[4] because it was portable, reasonably simple, not particularly controversial in the community (having several GUIs creates confusion and competition that is not always polite), and relatively easy to install. That "relatively easy" can be hard for someone who has never downloaded and installed a library before. Obviously, we install a version for students who use university computers, but more adventurous students sometimes have to be helped by the TAs. Setting up the correct compiler / IDE settings to use the GUI can also be quite frustrating.

Other software is just standard libraries or header-only libraries which cause no significant problems. To simplify the earliest classes, we use a header file that includes the necessary standard headers and an error() function; that way, we don't get embroiled in discussions about header files, system interfaces, and namespace management until those topics natually fit into the sequence of topics (e.g., Chapters 6, 8, and 12)

Non-problems: It is widely believed that any teaching of C++ is associated with endless searches for "buffer overruns" and misused pointers. We relegated such problems to the status of a minor nuisance through the systematic use of the standard library (in particular, range-checked vectors and strings) and the restriction of the use of pointers and arrays to the innards of classes.

8 Etc.

The ideal CS curriculum consists of so many topics that are fundamentally important, interesting, and in high demand by "consumers" (that is, industry and grad schools) that no student can complete it in four years – especially not if they take a reasonable load of humanities and science courses that they need to grow as individuals and to interact with non-CS people in the workplace. There is a serious information overload. My suggestion is to make a Master's degree the first degree considered suitable for a software development job. This view is hardly revolutionary: that used to be the view in Bell Labs and is the traditional view in many European countries. However, in a US university, this would be a culture change and is beyond the scope of a single department or a single university. Independently of that, we need to increase the level of professionalism, rather than pandering demands for

- "better trained" (but not educated) students from industry,
- "easier and more exciting courses" from students,
- "things done the way we are used to" from teachers unwilling to face a serious challenge,
- "more concrete, simpler-to-grade material" from TAs selected for their scholarly abilities rather than their software development skills or teaching experience,
- "more advanced / scientific / theoretical courses" from professors wanting Ph.D. students.

I am no fan of monocultures. I consider it essential that universities offer a variety of undergraduate courses with widely differing emphases in the area of software (and within other areas of computer science). Ideally, different software programs will emphasize different levels of "the software stack" and target different "consumers" of graduates (e.g. with emphasis on different programming languages and different application areas). Within the TAMU undergraduate CS curriculum we cater to the need for diversity of subjects by giving the students a choice of "tracks" with "software" being just one alternative.

Our emphasis on code, correctness, and classical CS topics may seem to go against the obvious and necessary aim of attracting more students to the computing profession. I don't think it does. It has been suggested that such an emphasis especially discourages women and minorities from studying computer science. On the contrary, offer anybody something they perceive as valuable and they'll come. Medicine did not go from having essentially no women to almost 50% women by offering cuddly soft choices. It did so by offering solid knowledge, professional status, good careers, and an obvious way for an individual to benefit society. Our freshman programming course contains a repeated emphasis on the benefits to society provided by software developers and explanations of the wide variety of applications areas. For freshmen, appeals to idealism are fortunately still more effective than quoting salary statistics (though the stellar statistics for CS, CE, and EE majors also help). Other courses and lecture series follow up on this. Whether that can achieve anything against "Hollywood"'s persistently negative image of programmers and engineers is doubtful, but it is a start.

9 Acknowledgements

Thanks to Walter Daugherity, Jaakko Järvi, Teresa Leyk, Ronnie Ward, and Jennifer Welch for constructive comments on drafts of this note and / or discussion about the first programming course. They all teach parts of the TAMU CS software curriculum. Also thanks to the WCCCE reviewers for constructive comments and hard questions.

10 References

[1] *Standard for the C++ Programming Language*. ISO / IEC 14882-2003.

[2] B. Stroustrup: *The C++ Programming language*. Addison-Wesley 2000. ISBN 0-201-70073-5.

[3] B. Stroustrup: *Programming – Principles and Practice using C++*. Addison-Wesley 2008. ISBN 978-0321543721.

[4] FLTK: *Fast Light Tool Kit*. `http://www.fltk.org/`.