Tutorial-Based Teaching of Introductory Programming Classes[1]

Joseph L. Zachary
zachary@cs.utah.edu
Department of Computer Science
University of Utah

In a typical programming class, students watch and listen as instructors explain and demonstrate key concepts through the use of blackboards, overhead projectors, and computer screen projection. Students then move to computer laboratories and solve programming assignments, more or less on their own. There is little opportunity for machine interaction in the lecture hall, and little opportunity for human interaction in the laboratory.

In the Department of Computer Science at the University of Utah, we have made substantial progress in merging the lecture and laboratory experiences into something greater than the sum of its parts. We have developed a computer-based learning environment and a set of online tutorials for teaching introductory courses in Unix, C, and Fortran. We have been teaching our Unix course using this technology for three years, and our Fortran and C courses for two years. In all three courses, which have served over 1700 students during this period, we have completely eliminated the lecture component. Student evaluations for all three courses are now higher than they were under the old lecture format.

During the summer of 1991 we developed software and courseware for teaching Unix to science and engineering undergraduates. We incorporated these developments into our curriculum beginning with the 1991–92 academic year. During the summer of 1992, we built upon our Unix experience by developing courseware for teaching introductory courses in Fortran and and C. We incorporated this work into our curriculum during the 1992–93 academic year.

1 A Tutorial-Based Unix Class

Introduction to Unix is a one hour course offered on a pass/fail basis. The course was created in response to the proliferation of Unix workstations across campus. The students who take the course are interested in learning enough about the workstations so that they can use them to complete work in other courses. The Unix course covers X windows, the Unix C shell, electronic mail, and Emacs.

Prior to September 1991, this Unix class was taught with one hour-long lecture per week, periodic computer-based assignments, and in-class written exams. We realized in the summer of 1991 that this was not a very effective arrangement. The lectures were entirely passive experiences, and consequently the students had trouble transferring what they witnessed in the lecture hall to the computer laboratory. While computer-based assignments certainly made sense, written exams on how to use a computer seemed the height of absurdity.

We considered the problem and realized that our Unix course was an excellent candidate for delivery via computer-based instruction. To understand why, consider how a teacher working one-on-one with a student, probably the ideal teaching arrangement, would operate.

- The teacher would explain the material carefully.
- The teacher would design and carry out demonstrations to reinforce the explanations.
- The student would practice the material just learned via hands-on experimentation.
- The explanations, demonstrations, and practice would lead the student to ask questions, which the teacher would answer.

We decided to approximate this process by replacing the role of the teacher with a combination of interactive computer-based tutorials and teaching assistant support. In our model, the student uses a workstation to run a tutoring program as one application in the X windows environment about which the student is learning. We will shortly describe the tutoring program in more detail. Notice the parallels with the ideal process outlined above.

- The tutor explains the material via structured text that the student reads.
- This structured text contains embedded buttons that, when clicked by the student, run simple demonstrations in the other windows of the display. For example, a button might illustrate a Unix command by running that command in an XTerm window.
- When using the tutor the student is, of necessity, sitting in front of a workstation. He or she is able to experiment immediately with the material at hand.

[1] This work was supported in part by Motorola Inc.

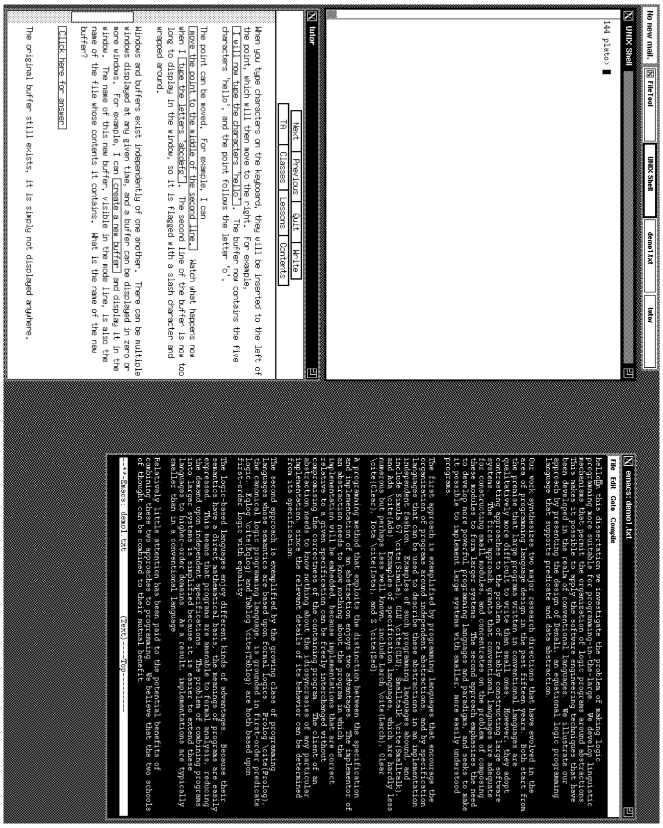

Figure 1: Workstation display of a typical Unix tutor user

- The workstation labs where the tutor is used are staffed by teaching assistants whose jobs are to answer the questions of tutor users. As we'll discuss below, the tutor software makes it easy to summon a TA for help.

This straightforward approach to teaching has proven to be extremely effective and has received overwhelmingly favorable reviews from our students. There are no lectures and there is no required textbook. We believe that the tutor is more effective than either lectures or textbooks because the student who is using the tutor is sitting at a workstation and becomes engaged with the material. A side benefit is that each student can work at his or her own pace.

1.1 Class Organization

The tutor is an application that runs in its own window under the X windows system. To understand how the tutor operates, it is helpful to know what a student sees when using the tutor. Figure 1 is the (rotated) X window display from the workstation of a typical student. The tutor is the window in the lower left corner. The lesson that is displayed in the tutor is an introduction to the Emacs text editor.

At the top of the tutor window are two rows of buttons. "Classes" allows the student to choose among different classes of lessons (currently Unix, Fortran, and C) and "Lessons" allows the student to choose one of the lessons that make up a class. "Contents," "Next," "Previous" allow the student to move through the sections of a lesson; "Write" saves a lesson section to a file; and "Quit" quits the tutor.

"TA" allows the student to summon a TA for help. Students who take the Unix class are supported by a team of teaching assistants. These TAs have two primary functions: they answer questions that arise from the material covered in the lesson, and they perform lesson checkoffs. To receive credit for a lesson, a student must demonstrate proficiency in the material to a TA, who then enters a password into the tutor, which keeps track of the student's progress.

The tutor maintains a central queue of TA help requests. The TA monitors the queue and serves students according to the order in which they request help. The student does not need to leave the workstation to find a TA, and the TA does not need to circulate through the building looking for someone who may need help.

Each student is free to work on any of over 80 workstations housed in two rooms on different floors of the engineering classroom building. Although students work at their own pace, there are weekly due dates to encourage regular progress. Students typically invest between two and three hours per week to complete the course.

The bulk of the tutor window is taken up by scrollable text containing boxed buttons. In Figure 1, the student has previously opened an Emacs window, which appears on the right side of the display. The tutor text describes various aspects of Emacs. The embedded buttons, when clicked by the student, illustrate the text by carrying out editing operations in the Emacs window. The student can also type commands directly into the demonstration window. The advantage of the buttons is that they allow the student to see what to expect from a particular command before using (or possibly misusing) it directly.

The current version of the Unix course involves completing the following nine lessons over the course of a ten week quarter:

Welcome	Emacs	Advanced Unix
Overview	Communications	Advanced Emacs
Unix Files	Unix Commands	X Windows and TWM

There are, of course, other on-line sources of information for all of the topics covered by the tutor. Emacs, for example, is documented by Unix man pages, the Emacs info mode, and an Emacs tutorial. Nevertheless, we believe that our tutor represents a unique and valuable addition to these resources. Three important reasons are:

- Online information sources such as the Unix man pages and the Emacs info mode provide too much detail. It is often difficult for a beginning student to know where to start looking, and it is also difficult to figure out what is important and what can be ignored. In contrast, we have designed the tutor lessons to give a high level overview and to get students started quickly with a well-chosen subset of available commands.
- The technical cast to the writing style of online documentation is often intimidating to beginners. We have written the tutor lessons in a conversational style.
- It is not always easy to learn how to use online information sources. It is easy to get lost, for example, when using the online Emacs tutorial since using it involve using Emacs. Our tutor provides a simple, uniform interface to tutorial material on a variety of topics.

1.2 Software and Courseware

In our workstation lab we routinely use the tutor on HP workstations running BSD and HPUX, Sun workstations running SunOS, and Motorola Delta88 machines running SystemV. The tutor, which is written in C++, ports easily to any Unix machine with a C++ compiler.

The variations among different machine architectures, versions of Unix, installation defaults, and (most of all) individual user customizations make it difficult, if not impossible, to write lessons that apply equally to all users. This problem has been exacerbated by our heterogeneous computing environment, and has been an obstacle to exporting our lessons to other installations. Over time we have adopted a variety of strategies for dealing with these variations.

Unix users can customize the appearance and behavior of almost every available utility through the use of "dot files." With dot files, users can do such things as create new names for existing commands in Unix, change the menus available from the window manager, change the appearance of windows, or modify the functionality of Emacs. New accounts are typically given a set of default dot files that vary from installation to installation. As a result, it is impossible to know with certainty how any given user's account will behave.

We have addressed this problem by creating a standard set of dot files with respect to which the lessons are written. New tutor users are encouraged to adopt these dot files; in fact, the tutor offers to do all of the necessary setup. Many of the lessons check to make sure that the user's dot files are the same as those expected by the tutor, and advise the user if they are not.

We have avoided the temptation to make the default dot files too specialized, because we want our students to be able to use Unix systems elsewhere. Rather than try to paper over the differences between different vendors' versions of Unix through the creative use of dot files, we have instead written our lessons to the lowest common denominator. We usually leave out features that are available on one kind of machine but not another.

Of course, it is not always possible to simply ignore the differences. A case in point is the standard Unix mail program. The sense of the two reply commands—'r' and 'R'—are reversed on our Sun workstations relative to our HP workstations. In this case, the lesson on mail explicitly tells the students about the machine dependencies.

We have created one set of lessons for all of the machines in our workstation lab, but they are not consistent with the versions of the lessons used at other installations. The tutor has been used in other workstation labs on our campus as well as at Brown University. In these cases, local changes have been made as necessary to the text of the lessons. As the lessons are written in a subset of LaTeX, these changes are easy to make.

1.3 Experience

The tutor software and courseware was originally developed during the summer of 1991 and has evolved considerably since then. We have offered our Unix class six times since the development of the tutor, and approximately 900 students have taken it.

During the last two years our approach to writing lessons has changed considerably. The first lessons that we wrote relied heavily upon demonstration buttons. A typical Unix lesson, for example, would contain text explaining a particular command followed by a button which, when clicked, would carry out the command in a demonstration window. It was often possible for a student to get all the way to the checkoff section of such a lesson without ever directly typing a command. Ironically, this tended to encourage the passive style of learning that we were trying to avoid.

We have taken a different approach in lessons that we have written or revised more recently. We use demonstration buttons much more sparingly, instead requiring the students to experiment directly. We also break up the explanations by asking the student to solve a problem or predict the outcome of a command. This style of lesson writing takes better advantage of the strengths of our approach.

We staff the workstation labs with TAs for 40 hours/week during a typical quarter, although the staffing level varies somewhat according to enrollment. We hire advanced undergraduates as TAs, with each TA working for approximately 10 hours/week. The staffing schedule includes daytime, evening, and weekend hours.

During the first two offerings of the Unix class we had trouble getting students to spread their work uniformly across the quarter. Each lesson had a due date, and many students would elect to complete the lesson on the day that it was due. Other students, realizing that the due dates weren't enforced, put off most of their lessons until the final two or three weeks of the quarter. These two forms of student procrastination fostered a poor learning environment, led to long waits for TA help, and ultimately burned out the TAs, which required training new TAs each quarter.

We eventually solved this problem by making two changes. We divide the class into five sections so that one fifth of the students have lessons due on Mondays, one fifth on Tuesdays, and so on. This tends to spread the student load evenly across the week. We also enforce the due dates by giving each student a "budget" of late days to spend as he or she chooses. Each student is responsible for completing all of the lessons before running out of late days.

2 Tutorial-Based Programming Classes

During the summer of 1992 we adapted our approach to teaching Unix for use in teaching our introductory Fortran and C programming courses. This entailed further software development on about the same scale as for the Unix course, but the courseware development effort was an order of magnitude larger. We introduced our work into the curriculum during the 1992–93 academic year.

Programming in Fortran is a three hour course and Programming in C is a four hour course. These courses are taught as a service for the rest of the University, as neither one is required for, nor generally taken by, computer science majors. A number of departments in the Colleges of Science and Engineering do, however, require their students to take one of these courses.

Prior to September 1992, both Fortran and C were taught using the traditional large lecture format, generally meeting in lecture three times per week. The classes also met in smaller discussion sections, led by teaching assistants, for one hour each week. Student evaluations were relatively low, though this varied considerably depending upon the instructor. We decided to try to duplicate what we had done with the Unix class and teach both classes with interactive tutorials, eliminating lectures and discussion sections entirely.

This was a far more radical idea than the previous year's, and in retrospect it is clear that we didn't appreciate how ambitious it was. We were setting out to prepare lessons for seven times the credit hours in the same amount of development time (one summer). Whereas the Unix course is a one hour credit/no-credit course, the programming courses are full courses in all respects. Neither of the programming courses require much in the way of prior programming experience. The Fortran course has only a college algebra prerequisite, while the C class additionally requires "some" programming experience.

Despite these reservations, we were (and remain) convinced that lecture classes are the wrong way to teach programming. As we had realized the previous year, lecture classes—especially large lectures with more than 70 students—are entirely too passive for teaching what is essentially a technical skill. In such classes the lecturer does most of the thinking; the students typically keep themselves busy taking notes. Our approach puts the students in front of the computer and keeps them busy reading and writing programs.

2.1 Class Organization

From the beginning, we realized that the programming classes would have to be run in a distinctly different fashion from the Unix class. The three most important distinctions involve the role of textbooks, the demands upon student time, and the means by which we evaluate and assign grades to students.

First, we have never required a textbook in the Unix course, choosing instead to make the on-line lessons complete and self-contained. This was not practical with the programming course, since we did not have nearly enough time to write what would have amounted to two complete on-line textbooks. Instead, we chose to require a textbook and then write lessons to complement it with examples and commentary, in much the same way that conventional lectures complement textbooks. We have used *Fortran 77 Programming* [Ell90] and *Applications Programing in ANSI C* [JK93].

Second, the Unix class is entirely self-paced; we require only that the students make steady progress and complete each lesson by the end of the quarter. This works because the class requires a relatively small time commitment from each student. We developed a different model for running the programming classes. We partitioned off some of our undergraduate lab space to create a dedicated electronic classroom, equipped with workstations. We divide each programming class into a number of 25-student sections. Each section meets twice a week for two hours at a time. There are two TAs present in each section at all times to answer questions and to monitor students' progress. During each section, students use the tutoring system to complete the lessons assigned for that day. Thus, while the individual lessons are self-paced, the course as a whole is not.

Finally, students in the Unix class are graded solely on the basis of whether or not they finish, to the satisfaction of the TAs, a sequence of lessons. This binary grading scheme works because the Unix class is graded on a credit/no-credit basis. The Fortran and C classes, with their letter grades, are managed differently. Each lesson ends with a checkoff problem—typically a small programming exercise—that the student is expected to complete during the class period. As in traditional programming classes, there are weekly programming assignments to be completed outside of class. Five times at regular intervals during the quarter there is an in-class programming quiz, which requires each student to solve a programming problem on-line by the end of the class period.

2.2 Software and Courseware

The students in our programming classes use a workstation environment identical to that used by the Unix students. We do not expect the students to have proficiency in using Unix workstations before beginning the class, nor do we try to teach them everything covered in the Unix class. Instead, we embed the details or what they need to know—directories, editing, compilation—within the lessons on C and Fortran. The students learn about Unix on a need-to-know basis.

Each lesson is coordinated with assigned reading from the textbook that the student is to have read ahead of time. A typical lesson builds on this reading by explaining the difficult, interesting, and key concepts from a new point of view. The explanations are illustrated by program examples, which students are directed to read, run, and modify. Each example program is designed to illustrate a single concept. Each lesson ends with a checkoff assignment that involves involve writing a small program that makes use of the ideas introduced in the reading and the lesson.

We have been able to support the Unix course, with its relatively low demand upon machine resources, with existing College of Engineering Sun and HP workstations. Offering the revised Fortran and C courses required dedicating 25 workstations for up to four days per week. We could not have done this without the generous support of Motorola, which equipped our electronic classroom.

2.3 Experience

The new software and courseware required for the Fortran and C courses were developed during the summer and fall of 1992. We have offered the Fortran class five times and the C class four times since then. There have been approximately 550 students in 23 sections of Fortran and 275 students in 13 sections of C.

Quarter	Fortran Rating	C Rating
Fall 1990	-1.37	-3.99
Winter 1991	-0.50	
Spring 1991	-0.84	-0.42
Fall 1991	-1.58	
Winter 1992	-1.71	-0.49
Spring 1992	-0.85	-0.97
Fall 1992	0.13	0.14
Winter 1993	0.51	
Spring 1993	0.27	0.34

Figure 2: Comparative Evaluations

Student evaluations reveal that the TAs have been the most highly valued aspect of the two courses. This is surprising because the majority of the TAs have been foreign graduate students. In past offerings of these programming courses, where the TAs have been required to lead discussion sections, foreign TAs have received extremely low evaluations from the students. It would appear that our TAs are much more effective in the one-on-one interactions typical of our tutorial-based approach.

Evaluations for the C classes are higher overall than those for Fortran. Recall that our C class requires prior programming experience whereas Fortran does not. One possible conclusion is that our approach may be better suited for students learning a second language than for students learning a first.

Course evaluations for Fortran and C covering the past three years are given in Figure 2. The ratings are a composite reflecting student satisfaction with the courses; the average rating is generally around zero, and ratings usually range from a low of -2 to a high of 1.5. Notice that the tutorial-based offerings (Fall 1992–Spring 1993) rate uniformly higher than the traditional offerings.

These evaluations are backed up by the anecdotal evidence of written student comments. Perhaps 5% of the students each quarter say that they would prefer a lecture-based class. Significantly, the students who have previously taken a lecture-based programming class are unanimous in preferring the current setup.

Not everything has gone entirely smoothly, of course. We have experienced problems having to do with hardware support, the programming environment, the lessons, and the TA workload.

We began our experiment in the fall quarter of 1992 by teaching a total of eight 24-student sections of Fortran and C using 24 NCD X terminals connected to a single Motorola MPC-300 machine. Each X terminal has a 68020 processor with four Mb of memory, runs the X server locally, and communicates with X applications running on the host machine via an Ethernet. Although the MPC-300 is fairly powerful, containing dual 88100 processors and 256 Mb of memory, it was over matched by the 24 X terminals. Furthermore, 24 terminals were not enough for 24 students, as it didn't account for occasional downtime or for students "visiting" a section for which they were not registered.

By the middle of the winter quarter of 1993 Motorola had provided an additional MPC-300 machine and ten more X terminals. Hosting 17 terminals on each machine produced more tolerable, if still somewhat sluggish, performance when the lab was full. And it was often full, despite the fact that we limited enrollment in each section to 25. We allowed students from other sections to work on homework sections on a space available basis while other sections were meeting.

We have written approximately thirty Fortran and thirty C lessons. The lessons typically describe some aspect of the language under study and direct the student to read, modify, compile, and run sample programs. Although the lessons have proven successful, there are other dimensions of student interaction that we would like to explore. For example, we would like to help students visualize the execution of programs by incorporating graphical debuggers or algorithm animation systems into the lesson mix. Much of what we currently do in the lessons could be done equally well in a textbook, and we would like to move beyond that.

Our experience has shown that the majority of each TA's in-class time is devoted to checking off assignments by looking at and running student-written programs to ensure that they are well-written and that they work properly.

There are two drawbacks to this labor-intensive approach to checkoffs. The first and most obvious is the drain on TA and student time. We would much rather see the TAs spend their time answering questions from students who are confused with the material rather than doing what is often a purely mechanical checkoff. Similarly, students do not always appreciate having to wait to show a TA that their program does in fact work. The second drawback is that students who are using the tutoring software but are not registered for the course, and who consequently do not have access to TAs, have no way to obtain feedback as to the correctness of their checkoff efforts.

We are working to make the software and courseware changes necessary to automate as many of the checkoff procedures as possible. Checkoffs that involve writing programs are the most common and, fortunately, are the easiest to at least partially automate. It would not be difficult for the tutoring software to run and test a program or module. TAs often give advice on programming style that would be invisible to an automated check. One solution may be to provide semi-automated checkoffs, where first the computer and then the TA must be satisfied.

References

[Ell90] T.M.R. Ellis. *FORTRAN 77 Programming.* Addison-Wesley, 1990.

[JK93] Richard Johnsonbaugh and Martin Kalin. *Applications Programming in ANSI C, Second Edition.* Macmillan, 1993.