Teaching Computer Programming Effectively Using Active Learning

Byron S. Gottfried
University of Pittsburgh

Summary

Over the past three years, we have learned how to provide effective instruction in computer programming within an active-learning environment. The use of active-learning does not in itself assure success in this area. However, we have found that we can provide effective instruction by utilizing a series of "mini-lectures" based upon carefully prepared examples that illustrate key features; by providing students with copies of the examples and encouraging them to write their own notes on the examples; by assigning simple in-class programming exercises that reinforce the material presented in the "mini-lectures;" and by supplementing the in-class activities with weekly programming assignments of a more comprehensive nature.

This paper describes each of these course characteristics in some detail. It also includes a list of features that work well, and another list of features, including some traditional teaching techniques, that we feel should be avoided.

Introduction

Ask most engineering graduates of the 1960s or 1970s what they remember most vividly about their undergraduate years, and they will probably recall, with nostalgia and some disdain, their experiences writing Fortran programs for a mainframe computer in a punched-card environment. During the 1980s, most engineering schools switched from mainframes to PCs and workstations, and some have adopted C or Pascal as their language of choice. At many schools, however, the *methods* used to teach programming have not changed, despite the dramatic changes in both the computing environment and the programming tools.

Experiences with Active Learning

At the University of Pittsburgh, all freshmen engineering students are required to take a course in C programming in their second semester, as a part of a common first year of study. Until three years ago, we had been teaching the course in much the same manner as in the 1960s and 70s: Two 50-minute lectures per week taught by a faculty member, and a two-hour lab per week taught by a graduate teaching assistant (TA). The lab session was largely devoted to answering questions (if there were any) about the weekly homework assignments. Classes were held in traditional classrooms, though the TAs occasionally took their students into a (somewhat underutilized) PC-based computer lab. Participating faculty were carefully screened, based upon their teaching ability and programming skills; some were experienced programmers,

distinguished teachers and authors of successful programming textbooks. Yet the student outcomes had been at best only mildly successful. Most students found the course uninspiring, despite concerted efforts by the faculty to provide interesting, high-quality instruction.

This began to change three years ago, when we decided to adopt an active-learning approach to both of our required first-year engineering courses. The traditional schedule (two 50-minute lectures and a two-hour recitation) was replaced by two 2-hour sessions per week, taught in a computer-equipped classroom (one student per computer). Each faculty member was now responsible for the entire four contact hours per week, assisted by a TA who helped individual students during each class session and graded the weekly assignments.

During a typical 2-hour session, the first 20 to 30 minutes were used for formal instruction (lecturing, which we refer to as the "L- word"), and the remaining time used for student hands-on exercises. Sometimes the mode of instruction shifted back and forth between formal instruction (lecturing) and active student involvement. We were very optimistic that this new mode of instruction would significantly improve the course.

During the first year, however, we were only modestly successful in improving the course. In our enthusiasm to bring about change, *we had included too many loosely coupled exercises that students found largely unrelated to one another*. (Some examples: Download a sample program and run it; change a program segment; run a program one way, then run it another way; find what's wrong with this program; etc.) We were lacking a central focus, to hold each day's instruction together. We later learned that *this central focus is an essential component of an effective active-learning experience*. Thus, we found that there is more to a successful active learning experience than a series of busy-work activities in a computer-equipped classroom.

In the second year we concentrated on the style of delivery and began to realize some significant improvements in the quality of the students' learning experience. We began by preparing a series of overhead transparencies that illustrated key points, largely by providing skeletal outlines of programs that served as prototypes for later student activities. Each student was provided with a printed copy of the entire transparency set.

Only three or four transparencies were shown each day, though the discussion of the transparencies frequently occupied 20 to 30 minutes of class time. Key points were pointed out to the students, and they were told to make notations on their copies of the transparencies ("*Write this down*!"). The students responded very positively to this - they seemed to appreciate the explicit emphasis on important key points.

At the conclusion of each day's transparency-based "mini-lecture," the students were assigned one or two simple programming assignments that made use of the concepts introduced earlier. The students were required to hand in a printed copy of each day's programming activity, primarily for accountability rather than grading purposes. A more comprehensive programming assignment, to be completed outside of class, was also given each week.

The Transparency Set

Over the past two years we have refined the transparency set, which now includes 96 separate transparencies. These transparencies serve as the backbone of the course. The nature of the transparencies can be seen from the representative examples shown in Figs. 1 through 3. Figure 1, for example, shows a very simple but complete C program. It is introduced during the second class session to illustrate several points, including overall program structure, symbolic constants, declarations, data input (with prompts), assignment, data output, and a representative dialog generated when the program is executed. A complete in-class discussion of this example requires 10 to 20 minutes.

Figure 2 presents a skeletal structure of a *for* loop, in which the number of passes through the loop is known in advance. This example is introduced during the fifth week of class. It illustrates several features of *for* loops, including the inclusion of an index, assignment of an initial value to the index, specification of a continuation condition, and a provision for altering the index at the end of each pass. About 10 minutes are required for a complete explanation.

Figure 3 shows a simple program that utilizes a subordinate user-defined function. This example is introduced in week 10 or 11. It illustrates a situation in which a numerical value is transferred from *main* to the subordinate function. This value is processed within the subordinate function, and another value, generated within the function, is returned to main. Roughly 10 minutes are required to fully explain this example.

Things That Work Well

In the past three years we've learned that the use of a carefully designed transparency set is only one of many different factors that contribute to a successful active-learning programming course. The factors that work well and contribute positively to the learning experience are summarized below.

- Present a *brief*, informal mini-lecture (20 to 30 minutes) each day.

- Use carefully designed transparencies to illustrate basic concepts (not more than 3 or 4 transparencies per session). The transparencies can include simple programs, skeletal outlines of programming constructs, or summary sheets. However, they must contain *meaningful information* (students will not pay attention otherwise), though they cannot contain so much information as to be overwhelming.

- Provide the students with *printed copies of the transparencies*. Have the students write their own notes on the printed copies to emphasize key points.

- Assign *in-class programming exercises* that reinforce the concepts presented in the transparencies. Begin with simple exercises, but assign additional, more challenging exercises to brighter students, who will finish early.

- Assign *one student per computer* during each class session.

- The professor and TA should wander around the class during the in-class programming exercises, *assisting individual students as required*. At the same time, *students should be encouraged to help each other*.

- Have students hand in a printed copy of their programming exercises at end of each session, mainly for *accountability purposes*. Do not grade them closely, but give each student a few points for successful completion of each exercise (or consider penalizing them for exercises not completed successfully).

- "*Slip in" new concepts informally*, before the concepts are formally discussed. This helps the student understand the concept intuitively, and it provides some constructive reinforcement when the concept is later introduced formally.

- *Assign reasonably comprehensive weekly homework assignments*. Grade them carefully and base a substantial portion of the course grade on the homework assignments. Show engineering relevance wherever possible.

- *Include at least two traditional exams*. Look for disparities between excellent performance on the homework and poor performance on the exams.

- *Select a textbook that is easily understood and not too long*.

Things That Don't Work Well

Success has its price. We've also learned that some things should be avoided, such as:

- Traditional 50-minute lectures.

- Detailed programming examples written on a chalkboard, which students write down in their notebooks (with numerous mistakes along the way).

- Detailed programming examples written on a chalkboard, with printed copies handed out to students. Though this works better than having the students write the programs by hand, it still blurs everything together and does not focus on key points.

- Placing too much emphasis on syntax, ignoring overall program design.

- Placing too much emphasis on program design, ignoring syntax.

- Selecting a sophisticated textbook, containing lots of detail.

- Insisting that students do the homework assignments entirely on their own (many won't).

- Assigning ineffective in-class activity, which may include too many unrelated or weakly related individual activities.

- Having students double-up on computers. (One does the work while the other daydreams.)

- Introducing a second programming language at the end of the course. (This approach is politically popular at many schools, though it does not contribute to good instruction. Just when students are at a point when they can be given a more comprehensive, capstone-type programming assignment, they are confused and sometimes bewildered by the details of a second language. This actually detracts from their learning either language.)

Conclusions

The use of an active-learning environment does not automatically guarantee effective instruction in a computer programming course. However, there are several steps that can be taken to insure the success of the course and bring about a favorable outcome. These steps include the use of "mini-lectures" based upon carefully prepared examples that illustrate key features, providing students with copies of the examples and encouraging them to write their own notes on the examples, assigning simple in-class programming exercises that reinforce the material presented in the "mini-lectures," and supplementing the in-class activities with weekly programming assignments of a more comprehensive nature. In addition, some widely used traditional teaching methods, such as 50-minute lectures and the use of detailed examples written on a chalkboard should be avoided.

BYRON S. GOTTFRIED
Professor of Industrial Engineering and Academic Director, Freshman Engineering Program, Univ. of Pittsburgh.
Department of Industrial Engineering, 1048 Benedum Hall, Pittsburgh, PA 15261
bsg@engrng.pitt.edu

Figure 1

A Representative C Program

(*Page 6 in Notes*)

Most programs consist of three basic steps:

1. Enter the input data (known information)
2. Process the input data to obtain the desired output (the answer)
3. Display the output data

```
/* calculate the area of a circle */

#include <stdio.h>

#define PI 3.141593

main()
{

      float radius, area;

      /* enter the input data */
      printf("Radius = ? ");      /* prompt for input data */
      scanf("%f", &radius);

      /* process the input data -> desired output */
      area = PI * radius * radius;

      /* or
      area = 3.141593* radius * radius;
      */

      /* display the output data */
      printf("Area = %f", area);

}
```

Program execution:

```
      Radius = ? 3
      Area = 28.274309
```

Figure 2

Looping - the For Loop

(*Page 38 in Notes*)

```
#include <stdio.h>

main()
{
     int count, n;
     scanf("%d", n);   /* n is the number of passes */
     for (count = 1; count <= n; count = count + 1)   {
         /* do something (repeated n times) */
     }
}
```

The **for** loop is best suited to situations where the *number of passes through the loop is known in advance* (i.e., **n** is known before entering the loop).

Notes:

The *initialization* (**count = 1**) occurs at the *beginning* of the *first* pass.

The *continuation* condition (**count <= n**) is tested at the *beginning* of *each* pass.

The *index adjustment* (**count = count + 1**) occurs at the *end* of *each* pass.

Figure 3

Functions and Function Prototypes

(*Page 66 in Notes*)

```
#include <stdio.h>

#define PI 3.141593


float funct(float r);          /* function prototype */


main()
{
      float radius, area;

      printf("Radius = ");
      scanf("%f", &radius);

      area = funct(radius);    /* function access */

      printf("Area = %f\n\n", area);
}
```

```
float funct(float r)          /* function definition */
{
      float a;

      a = PI * r * r;

      return(a);
}
```

Note:

The function prototype is required because **`funct`** is accessed (from **`main`**) before it is defined.

Without the function prototype, the compiler could not interpret the function access.