Computer programming fundamentals for non-computer scientists

by DANIEL P. FREEDMAN and THOMAS PLUM
State University of New York
Binghamton, New York

BACKGROUND

Computing for Social Scientists is one of a series of introductory computer programming courses designed for specialists, this one for graduate level social science students. A wide spectrum of disciplines was represented including anthropology, sociology, psychology, history, economics, linguistics, and urban planning. The students also had a highly varied background in mathematics—from one college math course through advanced calculus; from strong statistics to no statistical background whatsoever.

Computing background was equally varied. Few of the students had any formal background in programming, but some had been users of packaged programs. Other students had worked for faculty members in a data entry capacity, and others had monitored computerized scientific apparatus.

Even though most of the students were from non-computing fields, we chose to give them a full introduction to the new approaches in computing called “structured programming.” We wanted them to be aware that programs can be written with the goal of being correct as a direct result of the way they are written, rather than being thrown together in hopes of not being too hard to debug. We did not delve into the technicalities surrounding structured programming—rather we followed the non-technical approach of the introductory textbooks: Structured Programming in PL/C (by Weinberg, Marcus, and Yasukawa), and Structured Programming in APL (by Geller and Freedman). This approach not only gives them a higher standard of precision in their own work, but also shows them that they can read other people’s programs for correctness—freeing them, as consumers of software, from dependence on haphazard testing of the programmer’s product.

This emphasis on program reading paid off in some unexpected ways. A mini-computer show was held on campus, scheduled during the class period, so there was no difficulty getting the whole class to attend the show. Both instructors attended the exhibition, to see the displays and to explain the machines to the students. Of course they were also interested in seeing how the students approached computer sales personnel. Now, for the first time, the students had a real opportunity to play the role of consumer!

A student asked the salesman to display the program for the output on the fancy CRT. The salesman tried to avoid doing this by explaining that the program was written in BASIC, a language with which the student was unfamiliar. She remained undaunted and said, if the language was called “basic”, she should be able to understand it. Finally, the salesman displayed the program, and with a few words of syntactic explanation, the student was able to read the program. After examining the program for about five minutes, the student discovered a potential bug. Given a certain input, the program would blow up. The salesman said that the critical input would never be given, and the student smiled knowing full well that she would have given that input.

The objectives and priorities of the course were designed with the audience in mind. We knew that the majority of our students were becoming computer customers rather than computer programmers, but the emphasis was still placed on creating usable programs. Sheltering and reliability were the key objectives. Sheltered programs are designed with the user in mind. As far as possible, a program checks all input data for suitability. If the input is not correct, the user is informed, and the illegal input is not processed. Whenever possible the program control should remain within the program, rather than depend on the larger system for error correction or recovery.

Ease of data entry was also an important concern. The philosophy was that the user should not be exposed to all the peculiarities of the computer. The programs were designed to do formatting, both of input and output, and an output which was difficult to read was unacceptable.

The programs themselves were to be structured and readable. The programs were to be self-documenting and self-explanatory. “Let the program speak for itself.” Each program submitted had to be correct, and the correctness had to be demonstrated through a combination of structure and rigid testing. In addition to the testing, each program was examined and read by a classmate before submission. Any program which did not reach acceptable standards was returned, corrected, and resubmitted.

METHODOLOGY

The course was taught by a team of instructors, both of whom had considerable experience in computer program-
ming and social science. Although the instructors' backgrounds did not cover the total range of student interests, most major areas were covered between the two.

The class met twice weekly for two-hour sessions. One of these sessions was a lecture presentation by one instructor; the other was a workshop. The format of the workshop varied. Some weeks the class would meet at the computing facilities and students would work on their programs and receive individual assistance. At other times, exercises would be presented, homework reviewed, and specific programs discussed. The workshops were extremely successful in teaching practical debugging technique, program reading, and error checking. They created a congenial, pleasant work atmosphere for free discussion and exploration of computer concepts.

In addition to the instructors' presentation, computer center staff provided supplementary information about our computing installation. This presentation introduced the relevant packaged programs, computer center personnel, and informed the students of the services of the computer center. Computer center involvement was motivated originally by a sad story.

One day, one of the instructors was walking across campus when he encountered a colleague from another department. The professor was unloading a full station wagon. The cargo was box upon box of punched cards. During the conversation which followed, the instructor learned that the professor had just returned from a special trip to the Midwest to pick up his data and programs. These were statistical packages which had been written for the professor at his previous university. Subsequent discussion indicated that the statistical routines were fairly standard, and were contained in the packaged commercial programs offered by our computer center. It further appeared that this professor was never told about the possibility of transferring the data and programs to tape for ease of transport.

After this sad experience, the instructor realized the importance of teaching future social science professors where to find information—who to speak with about what, and a general appreciation of the capabilities of the modern computing system.

LANGUAGE AND TEXT

It has been our experience that teaching beginners two languages avoids language dependence. Once a student has mastered two languages, the third or fourth is not so awesome. The vast number of computer languages combined with the high rate of academic mobility requires that computer-oriented scholars be capable of adapting to different computer languages, facilities and installations.

With these factors in mind, two radically different languages were chosen for instructional purposes—PL/C (the Cornell version of PL/I) and APL. These two languages present an excellent contrast. APL is an interactive language with good facilities for handling higher level data structures. Its workspace limitation of 32K places a limit on the size of the problem. This "natural" limitation is fortunate, because our version of APL does not lend itself to large applications.

PL/C is a subset of PL/I, enhanced with excellent error detection facilities. The usually intelligent choice of error correcting options, and meaningful error messages aids the student to overcome many of the annoying pitfalls often encountered by beginning programmers. Our experience has shown that students who have mastered PL/I have little trouble reading and mastering other popular batch languages.

The use of these languages is further enhanced by the availability of two texts, Structured Programming in PL/C by Weinberg, Yasukawa, and Marcus, and Structured Programming in APL by Geller and Freedman, specifically designed for introductory students. The elements of design, program structure, clarity, and rules of structured programming are clearly incorporated into the programming procedure. Throughout the texts, programs which are progressively constructed and modified present a clear picture of the programming process.

By reading the many programs used as illustrative material, the student makes the acquaintance of good programming practices and learns to recognize good programming techniques. This last point was extremely important for our specialized audience. As consumers of software products, professionals, regardless of discipline, should be able to evaluate relevant packages. Even if these products are free, they still require time, which is the professional's currency.

EXERCISES

Some exercises were designed to reinforce a healthy skepticism on the part of the student as a consumer. Students were asked to choose a program from the APL library which had some application in their respective fields, or was just of general interest. Students were then asked to execute these programs, as if they were naive users; that is, read and follow the directions. They were not supposed to correct or debug these supposedly tested programs but rather, report the experience. They were to evaluate the same properties we had been stressing throughout the course; i.e., reliability, clarity of documentation, ease of use, clarity of program, clarity of output, and user protection. The results were interesting to the students though not surprising to the instructors. The majority of the programs did not do what they said they did. Most of the documentation was inadequate if not incorrect, and a good percentage did not execute at all. Of those which ran, many would blow up on incorrect inputs, and most had no means of recovery. In some cases the function which was supposed to print out the instructions failed to operate. This exercise vividly impressed the students with a sense of responsibility toward the user.

Other exercises were designed to move the students into taking a familiar function or operation and writing a program to execute the algorithm. An example of the problems...
was assigned to read as follows: "Complete a program called ENVIRON which will print out the distributional environment for a given letter. (Assume that you already have a corpus stored in a vector called TEXT.)" By consistently moving from the familiar to the unfamiliar, a methodology of programming exercises was presented, and by the termination of the problem, all students had been exposed to many solutions to the same problem. Thus, they learned that design is a choice among alternatives, not the only solution to the problem. The importance of design was further emphasized by having students modify their code, to make them aware of the difficulty in trying to modify a poorly structured program.

Essentially, exercises were designed to acquaint the students with a wide range of computer applications. As social scientists from different disciplines and different interests within disciplines, some would be doing numeric calculations whereas others might be dealing with textual problems. We felt that regardless of their immediate interests, each group should know the computer's capability in both these areas. Even if a sociologist never dealt with textual material, he or she should be a knowledgeable consumer of software.

THE FINAL PROBLEM

Remembering that our students were specialists in the social sciences and taking a computer course as an excursion in their educational careers, we wanted the final assignment to be of relevance to their primary interests. Therefore, the last assignment was broadly expressed. The student was to choose a significant problem within his discipline which lent itself to a computer application and attempt to solve it. We did not expect the resulting analyses to be earthshattering, but we did expect them to be meaningful and interesting. We were not disappointed.

Final projects could be broken down into two major categories. Many students explored the computer center resources and found packaged programs which were useful in analyzing data they had been compiling. Some of these applications were of a standard statistical nature, where others were new applications of standard routines in an extremely creative manner. One group of archaeologists applied a mapping program to archaeological-geological data. They were attempting to discover trade and migration patterns based on the distribution of raw materials in certain locations through time. An examination of this new form of data representation led to the formulation of hypotheses which were subsequently tested through standard archeological fieldwork methodology. Many of the hypotheses were validated.

CONCLUSIONS

We found that teaching, rather than "exposing", introductory computer programming to noncomputer people is a time consuming task. This course, which we consider a success, required two instructors, with a heavy commitment. All student programs were carefully read and discussed by both teachers. Given a class of 26 people, this required at a minimum one full day per week. Workshops and lectures were attended by both instructors, a practice essential for continuity. Team teaching requires the full time commitment of both people rather than the half time interest of two teachers.

Teaching computer consumerism involves many of the same principles as teaching good computer programming. Be skeptical. Read programs. Ask questions, and most of all remember the words "I don't know" and "I don't understand."

BIBLIOGRAPHY

PART II

METHODS AND APPLICATIONS