A First Course in Programming

Doug Bell and Peter Scott
Computer Studies Department,
Sheffield City Polytechnic,
Pond St.,
Sheffield. England

Abstract

Many of us have grappled with the design of a first course in programming - what language to use, how to sequence topics, how to encourage practical experience, how to assess. This paper explains the rationale behind a course that has matured over several years, and that we feel is sucessful.

Introduction

Most computer science courses involve a first course in programming. The topics include systematic design, coding, testing and debugging. (See for example the ACM recommendations in Koffman.) In designing our course we had a paramount objective: at the end of the course a student should be able to demonstrate complete competence in developing programs. The emphasis is on good design, rather than knowlegde of a programming language. We aim for the situation where any one of the student group can, with confidence, tackle a wide range of programming problems drawn from information systems, scientific programs and computer games. The key to this objective, we believe, is to focus on the process of design and to provide for extensive practical work. The implication is that we aim to minimise worries about programming language syntax and features. We shall return to a discussion of these issues later in the paper.

Other issues in designing a course are:

- the programming language to be used
- sequencing of material
- assessment
- the text book

We discuss these matters below.

As to the amount of time that a course takes up, we find that our course works best at two hours lecture and four hours practical work eack week for ten weeks.

The Design method

Because of its critical importance we teach the design method first. the method we use is functional decomposition - sometimes called stepwise refinement - in conjunction with pseudo-code. Our reasons for choosing this method are:

> The initial teaching can be carried out using human activities as examples. e.g. crossing the road.
>
> It is applicable to all applications areas - information systems, numerical problems, games, real-time.
>
> It is widely used in industry and commerce
>
> The use of pseudo-code is like using a programming language, so that there is no large gulf in going from one to the other.

The pseudo-code notation uses sequence, if...then...else...endif, while...do...endwhile and procedures, so that, for example, the design of a program to process a file looks like:

```
initialise
while not end of file do
    read a record
    process record
endwhile
tidy up
```

where "process record" is a procedure call. Design proceeds by writing a simple statement of the overall algorithm and then by writing down the detail of the procedures that are employed. And so on.

To gain proficiency, the students are asked to do a large number of fairly small problems involving human-related activities like crossing the road, the action of a vending machine, how to play clock patience.

The students don't use the computer during this stage of the course. But the fun element of getting hands-on can be accommodated by asking the students to become familiar with facilities for creating, listing, and altering files and running existing programs.

The programming language

The main consideration here is to avoid getting bogged down in the details of the syntax of any programming langauge. To this end we use a small subset of a language. As long as the programming language has certain essential features, then any of a number of languages is satisfactory. Pl/I, Ada, Pascal, Modula-2 and even a version of BASIC with suitable extensions are all suitable, but Pascal is most readily obtainable on the writers' computer system.

The subset of Pascal consists of:

```
data structures -
  integer, char, real, arrays

control structures -
  sequence, if...then, if...the...else,
  while...do, procedures with parameters.
```

We simplify the syntax of Pascal by insisting that there is always a begin...end pair in structured statements.

This is indeed a severe subset of Pascal (which is quite a small language to start with). So why have certain features been omitted ? We do not include Booleans because they are used less often and can be simulated easily with single character variables. Certain control structures - repeat and case - are unnecessary. We don't teach precedence rules in arithmetic or logical expressions because expressions are seldom complicated and precedence rules are complex (and error-prone). Instead we stipulate that students always use parentheses to avoid ambiguity. The two different parameter types for procedures in Pascal often cause confusion. Since value parameters are essentially only a convenience, we teach only variable parameters. Type definitions are left until any subsequent course in data structures. The omission of type definitions causes no problems in passing arrays as parameters if conformant arrays are used.

Although we have used Pascal, the same approach to subsetting could be employed in any high-level language.

Of course, we recognise that there are specific drawbacks with Pascal. The syntax rules for semi-colons are tortuous and error prone. There is no access to random access files, no separate compilation and no data encapsulation. But a major strength of Pascal is that we can use it in later courses on data structures.

One major issue that we emphasize in our course is the local vs global data debate. We teach the avoidance of global variables and recommend instead the use of local variables and parameters. On occasion, of course, global variables do have to be used (in Pascal). But we can make the decision to use them a conscious one by insisting that every student program has the following structure for its main program:

```
begin
main
end.
```

This causes compilation errors if the student does not pass parameters explicitly. Similarly, we do not teach nested procedure declarations as they also encourage implicit data sharing.

Sequencing

We have already mentioned that the design method is treated first. The next objective is to ensure that procedures are introduced early. This is helped by the fact that they will already have been understood and used during pseudo-code design.

The programme is as follows. Each week, two topics are introduced, making a ten week course overall:

```
Algorithm Design
More on algorithm design
Some algorithm design case studies
Computers, algorithms and
      the programming language
Grammar
Calculations with integers
Local variables and parameter passing
Selection and repetition
Processing character data
A case study in design and coding
Readability
Debugging and testing
Systematic working
Arrays
More on arrays
Real numbers
Top-down implementaion
File handling
```

Notice firstly that, (in the programming language) local variables and parameter passing are taught before selection and repetition. This reflects their importance.

Secondly, notice how, after the basics of the programming language have been dealt with, there is an intermission in the introduction of new langauge features to enable the student to apply and understand what he or she has learned. Some of the sequencing of topics is arbitrary, and determined by local needs. For example, reals are left until late in our programme because scientific and mathematical programming is not the first priority.

Practical work

Needless to say, we believe that people can only become good at programming by extensive practice. We take the view that it is better to develop many, small programs than a few, large ones. They supply a greater variety of problem areas, more design experience and avoid getting lost in developing elephantine programs. It also encourages the view that large programs are merely collections of small ones (procedures). Besides, the problems specific to very large programs are the material for any subsequent course in, say, software engineering.

Typical of the program specifications are:

> Develop a program to display a box like this on the vdu screen:
>
> ```
> ***********
> *         *
> *         *
> *         *
> ***********
> ```

> Develop a program to find the largest, smallest and sum of a set of numbers keyed in at the terminal. The numbers end with the number - 10,000.

> Develop a program to look up the train fare between two towns whose names are entered as data from the keyboard.

We have already mentioned that in the initial weeks of the course design exercises on paper are accompanied by familiarisation with the available computer facility. Thereafter the principal practical activity consists of the development of programs (design, coding, testing and debugging). But practical work can involve other activities. We have found that studying other people's programs is instructive. This can be done in several ways. On way is to have prepared model solutions to problems. Another approach is to ask students to pass arround their solution for scrutiny. A fun method is to jointly solve a problem as a group using a chalk board.

Assessment

The problems of assessing programming skill are well known. Program development takes a long time and a computer facility. Thus it cannot easily be assessed in a 3 hour exam. An alternative is to issue assignments that are done by a deadline, in the students own time. But here it is impossible to measure how much time a student has spent, so that the trial and error, experimental approach of the slow student equals that of the proficient student. It is also difficult to avoid the worry of collusion. In such a situation students tend to concentrate exclusively on the assessed work, ignoring all other work. Further, they typically take it easy until a week before the deadline. Then they go mad, ignore all other work and saturate the computer facility.

Our approach is novel and aims to overcome most of these difficulties. At the end of the course we conduct a 3 hour unseen exam to assess the paramount skill of algorithm design. During 3 hours, 5 or 6 small algorithms can be completed. The second component of assessment is continuous. Students are presented with a continuous succession of programs to develop throughout the course. They are told nothing about which programs will be assessed, nor when the deadlines will be. We encourage them to work smoothly and calmly an all the programs in the programme and to save design, listing and input and output in a work book. Periodically we announce a deadline for selected items. The deadline is a mere 3 hours ahead, which allows them time to remove their work from their work book in an orderly way, but is insufficient time for a last minute big-bang approach.

Our scheme works. The students address many more of the assigments, and they do so in a systematic fashion. Peaks and troughs are avoided.

Books

Many books on programming are very thick because they go into the extensive detail about the features of a programming language. We feel that this diverts attention from the central tasks of programming and is our reason for using only a language subset. Very few books adopt our approach. Notable is McGregor, though it is weak on parameter passing. Books like these are cheap, easy to read and do not put readers off with their excessive bulk.

Summary and conclusion

The essence of this course is a concentration on design, the use of a subset of a programming language and extensive practical work.

****

FIRST COURSE-- continued on page 57

error detection and correction techniques to the existing software such as Hamming code and observe how this allows errors to be corrected successfully.

Conclusions

The students were not required to write software for these practicals, but were asked to make simple edits to the programs supplied. This gave them an introduction to the UCSD system. The values chosen for the simulated lines are the same as those used by British Telecom and nearest preferred values were used. The noise circuit board was designed to run at 110 baud in order that the line could be observed clearly on an oscilloscope. This was inconvenient since all the comms cards have printers attached running at 9600 baud, and had, therefore, to be set and reset before each session. I intend adding an interface chip onto the noise boards to overcome this.

References

1. W.A. Coey and D.Q.M. Fay. "Practical Computer Logic Classes for Computer Science Students: The Use of Logic Analysers". SIGCSE BULLETIN Vol. 14, No. 3, September 1982.

2. BS 6317 : 1982. Simple Extension telephones for connection to the B.T. public switched telephone network.

****

FIRST COURSE-- continued from page 50

We have succesfully run this course over a number of years for a variety of students - school leavers, and mature students. The course has been successful in enabling the students to become completely confident and competent at programming. Just as important, the courses have been fun.

There are, of course, casualties but those who pass the course are certain of a firm base of skill which they can use just as it is, or they can build upon the skill in later courses.

Most students nowadays have experience of programming before starting our course. But even though this course starts from scratch, these students find the different approach enjoyable and rewarding.

References

Koffman E B, Miller P L, Wardle C Recommended Curriculum for CS1, 1984. Commun. ACM 27, 10 (Oct 1984). 998-1001.

Moffat, D., and Moffat, P. Eighteen Pascal texts: An objective comparison. SIGCSE Bull. 14, 2(June 1982), 2-10.

Mcgregor J J, Watt A H, "Simple Pascal", Pitman 1981

****

DATABASE-- continued from page 54

15. Husnian, David, "Why dBASE III is the Best General-Purpose Applications Language," SIGSMALL/PC NOTES, Volume 11, Number 2, May 1985, p. 23-26.

16. Salton, G., "Some Characteristics of Future Information Systems," SIGIR FORUM, Volume 18, Issues 2-4, Fall 1985, p. 28.

17. Kleinrock, Leonard, "Distributed Systems," Communications of the ACM, Volume 28, Number 11, November 1985, p. 1201.

18. Byers, Robert A., Everyman's Database Primer Featuring dBASE II, Ashton-Tate, Culver City, California, 1982.

19. Mugridge, Warwick B., "A Method for Introducing Schemas," SIGCSE BULLETIN, Volume 17, Number 4, December 1985, p. 76.

20. Sacca, Domenico, and Wiederhold, Gio, "Database Partitioning in a Cluster of Processors," ACM Transactions on Database Systems, Volume 10, Number 1, March 1985, p. 30.

21. Polilli, Steve, "'Distributed' R/DBMS Beats IBM to Market," MIS Week, Volume 7, Number 23, June 9, 1986, p. 44.