A First Course in Computer Programming

using PASCAL

ARTHUR M. KELLER

COMPUTER SCIENCE SERIES

A FIRST COURSE IN COMPUTER PROGRAMMING USING PASCAL

Arthur M. Keller
Stanford University

McGraw-Hill Book Company
New York St. Louis San Francisco Auckland Bogotá Hamburg
Johannesburg London Madrid Mexico Montreal New Delhi
Panama Paris São Paulo Singapore Sydney Tokyo Toronto

This book was typeset using the TEX document production system, and camera-ready copy was produced on a CRS Alphatype phototypesetter. Computer resources were provided by the Stanford Artificial Intelligence Laboratory and the Stanford Computer Science Department. The editors were James E. Vastyan and Joseph F. Murphy; the production supervisor was Diane Renda. The cover was designed by Steven Hoffman.
Edwards Brothers Incorporated was printer and binder.

A FIRST COURSE IN COMPUTER PROGRAMMING USING PASCAL

6789 SMSM 8987654

ISBN 0-07-033508-7

Library of Congress Cataloging in Publication Data

Keller, Arthur M.
A first course in computer programming using PASCAL.

(McGraw-Hill computer science series)
Bibliography: p.
Includes index.
1. PASCAL (Computer program language)
I. Title. II. Series.
QA76.73.P2K44 001.64'24 81-23601
ISBN 0-07-033508-7 AACR2
ISBN 0-07-033509-5 (Instructor's manual)

To my father,
who taught me the importance of learning,

and to my mother,
who taught me the importance of not doing it all of the time.

Preface

This is a complete textbook for a first semester or two quarter course in computer programming using the PASCAL language. This book could be used at the undergraduate level or at the advanced high school level. It is compatible with the curriculum for the course **CS 1** as described in Curriculum 78.* The coverage extends beyond this curriculum to cover the entire PASCAL language.

There are several unique features of this book. Procedures are taught very early. This fits well with the technique of stepwise-refinement method of top-down decomposition style of programming. Many examples show several stages, some of which are correct and others incorrect. Examples of good programs are marked by the "thumbs-up" symbol that appears in the margin of this page. Examples of incorrect programs are marked by a "thumbs-down" symbol in the margin. In this way, the reader learns why programs are incorrect and how to debug programs. Each new topic is introduced by an example that illustrates the feature and explains the need for this new feature. In this way, the student learns to think from the point of view of solving the problem and determining what features are needed to solve a problem. After introducing a problem that motivates learning a feature, the feature is explained, and then the problem is solved in depth.

This book assumes no knowledge of higher mathematics on the part of the reader. The reader should, however, know elementary high school algebra. Although many explanations were devised for readers without mathematical backgrounds, those with mathematical sophistication should not be bored or insulted either.

There are different approaches to teaching an introductory programming course. The most common variations affect the order of covering topics. Some of the topics in this book may be covered in a different order if desired. The most important of these is the coverage of procedures and of procedure parameters.

Procedures are a very simple but important control structure. The concept of top-down programming requires the procedure abstraction. Consequently,

* "Curriculum '78: Recommendations for the Undergraduate Program in Computer Science" in *Comm. ACM*, **22**, 3 (March 1979), 147–166.

the first control structure covered in the text is procedure declaration and invocation. When students learn to program without being taught to decompose their programs into procedures, they often write long unstructured programs without using any procedures, even long after procedures have been covered. The main drawback with the approach of teaching procedures early is the use of references to nonlocal variables. Even when parameters are taught, students may still prefer to use nonlocal references. Introduction of procedures can be delayed until students have learned several other control structures. While this may facilitate utilization of this book with existing curricula, it only partially alleviates the objection raised. Alternatively, procedure (value) parameters may be taught earlier. Section 9.1 may be taught as early as Chapter 3. Value parameters should be taught at least a week before **VAR** parameters so that the students understand them before being introduced to the additional nuances of **VAR** parameters. Early coverage of value parameters allows such time, and it also solves the problem of nonlocal references. Value and **VAR** parameters were placed in the same chapter primarily to assist in comparing them.

Coverage of recursion may be delayed or omitted. Its placement was dictated by being a prerequisite of the merge sort. Coverage of merge sort may also be delayed or omitted. However, the merge bubble sort should be covered when the other sorting algorithms are covered, primarily because it is faster than n^2 and does not require recursion.

Chapters 16 to 21 may be covered in any order. These chapters complete the coverage of PASCAL, and their coverage may be delayed until the second course in computer programming.

An *Instructor's Manual* is available that includes brief discussions of approaches to using the text, extensive programming problems and solutions, and explanations of what is needed to solve the problems and how to go about solving them.

Acknowledgments

I have received assistance from many people while preparing this book, and for this I am extremely thankful. Rich Pattis provided many ideas and suggestions; his book* and notes were often a source of inspiration. I received much help from Dave Wall during numerous discussions. Jeff Vitter helped shape the presentation when we taught together at Stanford; he assisted further using the book in preprint form at Brown University. Brent Hailpern provided many useful suggestions for improving this book. Denny Brown gave extensive advice about pedagogy. I would also like to thank my teaching assistants who filled in the cracks in my explanations. I also thank my proofreaders for catching many of the bugs before they could confuse students. I will gladly pay a $1.00 reward to the first finder of every remaining error.

I would like to thank Stanford University, and especially the Computer Science Department, for providing an excellent learning environment and inquisitive students. The staff of the LOTS computer facility was very helpful in providing an environment where students can learn about computers and in correcting my technical misconceptions. The SAIL and SCORE computer facilities of the Computer Science Department were used to typeset this book and to test

* *Karel the Robot: A Gentle Introduction to the Art of Programming*, Wiley, New York, 1981.

all the programs used in the book. Much TEX wizardry was provided by Jim Boyce. The document compiler used, TEX, was designed by Donald E. Knuth. This book is typeset using the Computer Modern series of fonts (also designed by Knuth). Some of the graphic characters were designed by Scott Kim.

I would also like to thank the following people and organizations for their assistance in the preparation of this book: American Film Institute, Jim Arnold, Jim Celoni, S. J., Tom Dietterich, Les Earnest, Ed Feigenbaum, Robert W. Floyd, Martin Frost, David Fuchs, Richard P. Gabriel, Howard Givner, Ron Goldman, Gene Golub, Ralph Gorin, Lynn Gotelli, Susan Hill, Fran Larson, Frank Liang, Richard Manuck, John McCarthy, Jim McGrath, Mike Peeler, Jayne Pickering, Michael Plass, Stuart Reges, Betty Scott, Laurie Sinclair, Richard Southall, The Stanford University Libraries, Jorge Stolfi, Carolyn Tajnai, Chris Tucci, Jeffrey Ullman, Marilynn Walker, Gio Wiederhold, Don Woods, and Dawn Yolton. I also thank the McGraw-Hill Book Company for their extensive assistance. Finally, I would like to thank my students who made it all worthwhile.

Arthur M. Keller

Contents

Chapter 1

Introduction to Computing

In this book, we will learn how to get computers to do work for us. Even those students who will never program after reading this book will still learn what computers can do and how they work. Our problem-solving skills will be exercised and strengthened. In particular, the technique of problem decomposition will be learned.

A computer is an automatic tool. It is intended to do work for people, not to control them. It can add a column of numbers rapidly. It can make certain kinds of decisions, much like a thermostat which "knows" when to turn heat on and off.

A computer must be told how to do something. It naïvely follows instructions. We have to tell it step-by-step everything it is to do. These instructions comprise a *computer program,* which is similar to recipe in a cookbook.

Consider the following recipe for duck à l'orange:*

> This famous recipe depends for its flavor on the Seville or bitter orange, 135, which gives the dish its name.
> Prepare:
>
> **An unstuffed Roast Duckling, above**
>
> When it is done, remove it from the roasting pan and keep warm.
> Prepare:
>
> **Sweet-Sour Orange Sauce, 355**
>
> using Seville or bitter oranges and omitting the lemon. Degrease pan juices and deglaze the pan as described on 340.

* Irma S. Rombauer and Marion Rombauer Becker, *Joy of Cooking,* Bobbs-Merrill, New York, 1975, p. 433. Reprinted with permission of the publisher.

The first sentence tells something about the recipe. It says we can find information about the Seville orange on page 135. The first step—prepare an unstuffed roast duckling—relies on a more basic recipe shown earlier on the page. The next step is simple. After it, we reach a step described in more detail elsewhere—the preparation of the sweet-sour orange sauce. However, we prepare it differently than usual by using the Seville orange and omitting the lemon. The last step is explained on page 340.

In describing the solution to a problem, we use a layered approach. We first use a general outline of the basic steps. The outline is then elaborated with more detailed steps. Gradually, we flesh out the outline until everything is described in sufficient detail. This is also the time-honored method of writing an essay: after creating the detailed outline, we convert it into a complete paper.

Computers need to be told what to do—and how to do it—unambiguously. Unfortunately, English and other ordinary languages are too vague. Consider the sentence, "The lady made the robot fast." This brief sentence can have many meanings:

1. The lady built the robot quickly.
2. The lady designed the robot so that it would operate quickly.
3. The lady took a slow robot and speeded it up.
4. The lady tied down the robot.
5. The lady forced the robot to stop eating.
6. The lady attended a gathering of robots that were not eating. (Compare with "The lady made [it to] the robot *feast*.")

Rather than using English, special languages—*programming languages* or *computer languages*—have been developed. These languages are very precise, and enable the computer to interpret a program unambiguously. Just like the step in writing an essay of converting the outline form into English, there is a step in writing a program of converting the outline form into a programming language.

Once a program is written, it is not yet finished. An essay, once written, must be polished. The acid test for a computer program is to feed it to the computer. The computer takes the program and follows the instructions in it. This is called *running* or *executing* the program. The computer may have difficulty following the instructions. For example, the program may have an error in grammar. Since the computer will usually read what you tell it to do before attempting to do anything, grammatical errors are the first to be found. Once you have fixed all the grammatical errors, the computer will attempt to run your program. Your program may ask the computer to do something it cannot do. For example, you may have asked it to divide a number by zero. Such errors are usually harder to find and correct. We will learn techniques for figuring out what is wrong with one of our programs and how to correct it.

1-1 ALGORITHMS

An algorithm is a type of description of the solution of a problem. The recipe given earlier was an algorithm for making duck à l'orange. Other algorithms include knitting instructions, instructions for constructing a kit, and computer programs.

An algorithm is a vehicle for explaining how a problem can be solved. It necessarily uses a step-by-step approach. It can be formulated in a variety of ways, provided that it is unambiguous. Each field of study has developed its own specialized vocabulary for describing how things are done. Unambiguous languages, such as PASCAL, have been designed for use in describing algorithms to computers.

An algorithm must be precise. It must tell the order of steps. When putting together a kit, it is important to know whether to fit tab *a* into slot *b* before or after fitting tab *c* into slot *d*. A list of ingredients is not enough; we also have to know when to add them. An algorithm must also clearly tell when to stop doing something and to go on to something else. A recipe that just says to bake a cake until it is done is not as useful as one that also gives criteria for when the cake is done. Consider the following algorithm for making toast:*

There's an art of knowing when.
Never try to guess.
Toast until it smokes and then
twenty seconds less.

We do not know when to stop cooking the toast until it is too late! An algorithm has to describe how to choose between alternatives. When buying apples from a store, you cannot choose the tastiest apples because it is hard to know which apples are going to be the tastiest without actually tasting them. However, you could tell how tasty a bunch of grapes was by eating one (but the store might still not like it if you ate a grape before buying the bunch).

An algorithm must be definite. If you follow an algorithm twice, the same result must follow each time. Two people following the same recipe for pie should bake equally tasty pies. If this is not true, they must have used different brands of ingredients, varied the recipe slightly, or changed something else. Technically, we want an algorithm to be *deterministic.*

An algorithm must be finite. If you follow an algorithm, it must end eventually. Suppose someone else is thinking of an integer—it could be positive, zero, or negative—and we want to guess it. Consider this algorithm. Try 0. Then, try 1. Then, try 2. Continue with some more positive numbers. Then, try −1. Then, −2. Continue with some more negative numbers. This process is supposed to stop when we have guessed the number. If the other person is thinking of a negative number, we would never guess it because we would never run out of positive numbers to guess, and we guess all positive numbers before guessing any negative numbers. A better algorithm follows. Try zero. Then, try 1 and −1. Then, try 2 and −2. Continue trying positive and negative numbers until guessing the right number. The number of steps we take is about twice the magnitude (absolute value) of the right number. For example, we guess 10 in step 20 and −10 in step 21. Thus, we see that the later algorithm is finite while the former need not be.

The definition of an algorithm should describe three parts: input, process, and output. An algorithm usually involves some *input*, that is, things that exist that are used by the algorithm. The input for a recipe includes the ingredients and the utensils used. An algorithm also produces results called *output*. The output for a recipe is usually some tasty food. As we have already considered, an algorithm describes how the input is to be transformed into the output.

* Piet Hein, "Timing Toast" in *Grooks 2*, Doubleday, Garden City, N.Y., 1969, p. 23. Reprinted with permission of the author.

1-2 STEPS IN THE LIFE OF A PROGRAM

Only a small part of the time spent in the development of a program is actually in writing it. There are several other important steps.

The first step in writing a program is problem definition. If you do not know where you are going, you cannot know it when you get there. Often the problem we are asked to solve is ill-defined. The input or output may not be clearly stated. Consider the assignment of mailing the best customers an offer to try a new product. There are several important questions that arise. Where is the list of customers? How do we know which customers are the best customers? What should the offer look like and what should it say? These questions have to be answered before we can proceed to writing the program.

Once the problem is defined, we can outline the solution. We consider general alternative approaches to solving the problem. Our first solution may not be the best way of solving the problem. At this stage, we have not invested very much in any particular way of solving a problem. If we devise a better solution now, we will have saved more time than if we have to adopt it later. Consider the problem of finding a telephone number. Suppose we decide to look it up in the telephone book. We could search it sequentially, but on the average we would look through half of the names. Such an algorithm would be easy to describe but very slow. We could take advantage of the index entries at the top of the page by thumbing through the telephone book until we find the right page. Then, we can search the page sequentially. This algorithm is harder to describe, but will take much less time to follow. Of course, this case is clear cut, but with other problems the choice may not be as obvious.

The chosen outline is then developed into an algorithm. We will use a top-down approach to developing an algorithm. This means that we start with the outline and define each step in greater detail. This fleshing out continues until we are sure exactly how to solve the problem. To some extent, this involves using existing algorithms. For example, when a step of the outline is to do something we have done before or is in a book, we can do it the same way. As we develop more complicated programs, we will build up a repertoire of techniques that we can put together in new ways to solve new problems.

After the algorithm has been chosen, we proceed to writing it in a programming language. This step translates the abstract algorithm into PASCAL. Writing the actual statements in a programming language is called *coding*, since a sequence of such statements is called *code*. It is important not to do any coding until the algorithm is fairly well defined, lest we become committed to the code and not be willing to change our minds. Programming should not be confused with coding: coding is just one of the steps in programming.

When parts of the program are coded, they can be tested. This involves having the computer try to understand the program and then follow it. It is best to test the components of a program separately before putting it all together, especially for a large program. Otherwise, it will be hard to track down the source of the errors. Errors in a program are called *bugs* and detecting and correcting them is called *debugging*. The process of debugging involves testing the various cases the program is expected to handle. If we find any errors, such as incorrect output, we look for the cause of each error. For each error, we try to devise a change to the program that removes the problem. Such changes often remove other errors but occasionally introduce new ones, so it is necessary to retest the program to ensure that it still works for the old cases and handles the new ones properly. We continue testing the program and correcting errors until we have tested all cases and find no remaining errors. *Testing only shows the presence of bugs, not their absence.* There may be other bugs, but we have not found them, probably because we have not tested every possible case.

We have so far only considered the program, but not the explanation of the program for others. As we will find, a program can be quite complicated and confusing. Hence, it is important for the program to include explanations of what it is intended to do and how it works. These explanations are called *documentation.* In order to make programs more readable to others, several techniques are often employed. A program is formatted to reflect its structure—similar to the indentations in an outline that reflect its levels. Small discussions are sprinkled throughout the code to explain how the program works. These are elements of internal documentation, which are explained fully later in the text. A formal document—describing the input, output, function of the program, and how the program works—is written to enable the effective use of the program. Such documentation also makes modifying the program easier. Although there is some tendency to delay writing documentation until the program is fully debugged, it is best to write it during development of the program. The original specifications can become the basis for the external documentation. As the design proceeds, decisions can be incorporated into the documentation when they are made. Very little documentation should be left for the coding step or after debugging. Once the program is working, we have probably forgotten most of the design decisions. All that should be left is to polish the documentation, not to write it.

Classwork problems end there, but real programs continue to live on. A real program is written because there is some real problem to solve. Once the program is available, people may think of other useful things it could do, or cases it does not handle properly that were not clearly specified. This ushers in the *program maintenance* phase. Changes will have to be made to the program to satisfy the changed requirements for the program. Sometimes these changes will be made by the original programmer; other times they are given to someone else to do. If we are given a program to modify, we must first learn how the program works, and then figure out how it should be changed. For many programs, this process is more expensive than the original program development! Program maintenance can be facilitated by good programming habits, such as writing lucid code and providing good internal and external documentation.

1-3 COMPUTER ORGANIZATION

Learning about the organization of a computer will help us understand how a computer works and how to use it. The topology of a computer is like that of a star. The center of a computer does the work and makes the decisions, analogous to the function of the brain. Surrounding the brain are devices for communicating with the outside world. These include input devices, like sense organs, and output devices, like vocal cords. Some devices participate in both input and output, such as a computer terminal. The hands and mouth are also input and output devices.

The central part of the system is called a the *central processing unit* (CPU). This part follows the instructions of a program and directs the input and output devices. Some memory is included which contains, among other things, the parts of a program being executed and the information being manipulated by that program. This memory is often called *core memory,* originally because it was composed of tiny copper cores, but today the name continues to be used because it signifies the central or primary memory of the computer.

There are many varieties of input and output devices. A common output device is a printer. Printer speeds range from 300 lines per minute or less to more than 2000 lines per minute. Most printers print only fixed-width characters like a typewriter, and some cannot print lower case letters. Some newer printers can produce print graphics and diagrams; for example, this book was typeset on a graphics computer printer. Some computers have devices for reading and punching computer cards. The rectangular or circular holes in the cards is what the computer is interested in. The computer does not read any of the writing on the cards. The familiar warning against "folding, spindling, or mutilating" computer cards was written so that the cards can be fed into a card reader (at speeds upwards of 1000 cards per minute) and then read accurately. Customers are often instructed to write something on the card, but it is *keypunched* on the card before the computer is able to read it. An increasingly common device capable of both input and output is a computer terminal, also known as a CRT because the screen is often a cathode-ray tube. Unusual output devices include voice synthesizers and robot arms. Computers will soon be able to audibly warn a driver that a car is low on gas, or claim that it has just won a chess game. Unusual input devices include cameras and laboratory instrument sensors.

The average computer does not understand PASCAL directly. This is similar to the problem faced by the native speaker of English who also speaks French, but not fluently. That person thinks in English and translates everything heard or read from French to English and everything about to be said or written from English to French. A computer's native language is called *machine language* and consists of a binary code (numbers in base 2) containing only 0s and 1s. The first computers were programmed exclusively in machine language. A symbolic language that could be easily translated into machine language was soon developed; it was called *assembly language* because the machine instructions were assembled directly from assembly instructions using—you guessed it!—an assembler. Then, high-level languages were invented—the first was FORTRAN in the early to middle 1950s—and *compilers* were constructed to translate the programs to machine language. These languages are called high-level languages because the translation of programs written in them into machine language is complicated, and each statement in a high-level language may compile into several machine language instructions.

A compiler is a program that takes another program as input and produces machine language and messages as output. The input program is called the *source program* while the output machine language is called the *object program.* After the compiler is run, the resulting object program is run. This program does whatever is requested in the original high-level language program. It also has its own input and output.

1-4 GOOD PROGRAMMING

Good programming is an art, and opinions vary about the best way to write a program. However, three important criteria for determining whether a program is good are correctness, clarity, and efficiency.

A *correct* program does what it is supposed to. It conforms to its specifications, in that its output is correct for any acceptable input. The problem is that the program has to work for any acceptable input. *It is not possible to test every*