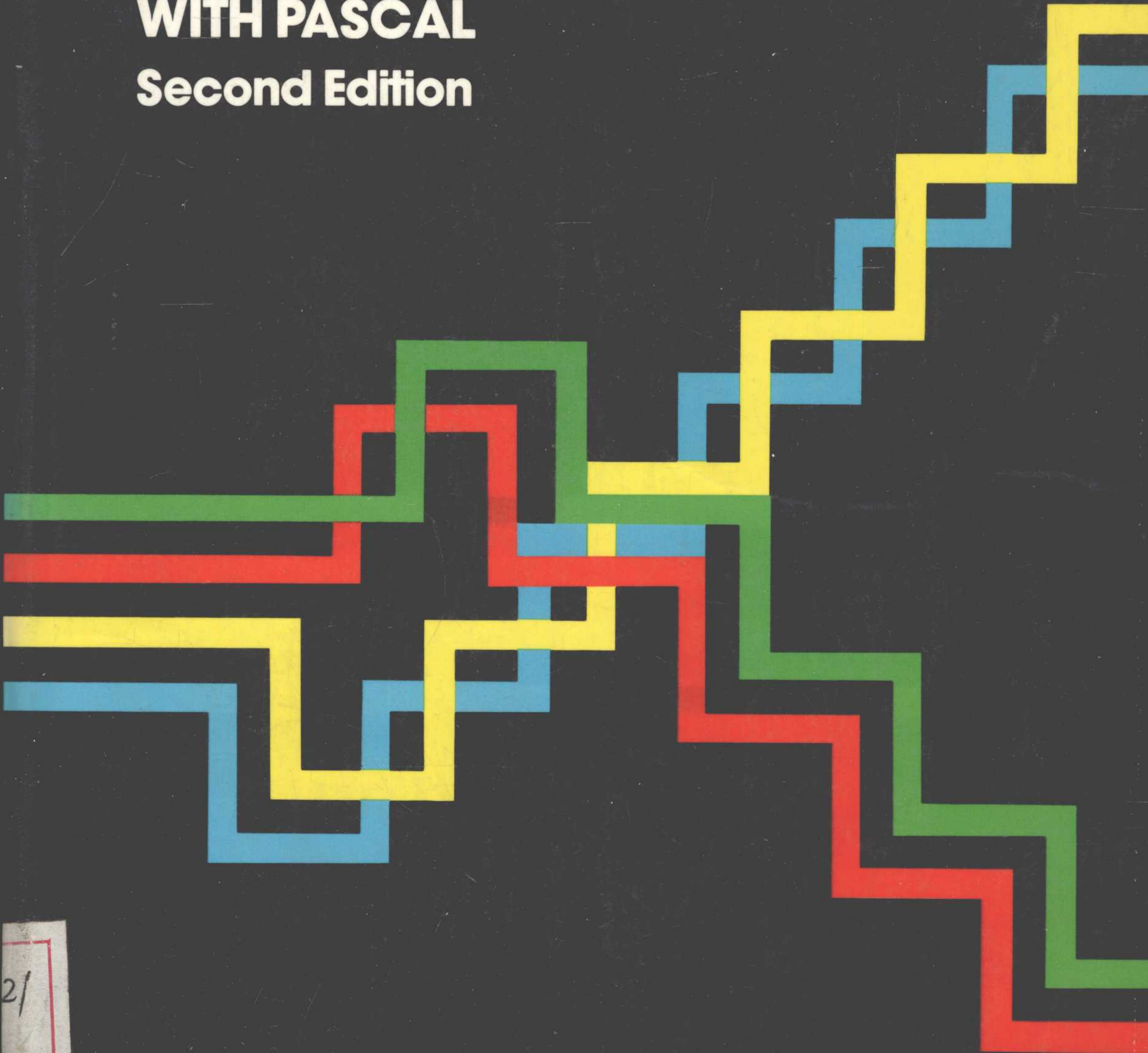
An Introduction to

PROGRAMMING AND PROBLEM SOLVING WITH PASCAL

Second Edition



G. Michael Schneider
Steven W. Weingart
David M. Perlman

G. Michael Schneider
Macalester College, St. Paul, MN

Co-authors of the first edition:

Steven W. Weingart
Data General Corporation

David M. Perlman
Cray Research

Second Edition

AN INTRODUCTION TO PROGRAMMING AND PROBLEM SOLVING WITH PASCAL

John Wiley & Sons
New York *Chichester* *Brisbane* *Toronto* *Singapore*

To my wife, *Ruthann,* and my children, *Benjamin* and *Rebecca*

Library of Congress Cataloging in Publication Data

Schneider, G. Michael.
An introduction to programming and problem
solving with PASCAL.

Bibliography: p.
Includes indexes.
1. PASCAL (Computer program language) 2. Electronic digital computers – Programming. I. Weingart, Steven W. II. Perlman, David M. III. Title.

QA76.73.P2S36 1982 001.64′24 82-2809
ISBN 0-471-08216-3 AACR2
ISBN 0-471-80447-9 pbk.

Printed in the United States of America

20 19 18 17 16 15 14 13

FOREWORD

The maturation of programming into an art or a science is characterized by our ability to abstract essential principles from particular cases. As fundamental concepts emerge, a notation is needed to express them. We have become accustomed to calling such a formal notation a ''language.'' The better it is tailored toward expressing the essential abstractions, the better it is suited to introduce the subject of programming, because the more it can recede into the background. Ideally, the language presents itself as the natural notation to express the basic concepts of programming, and it should hardly need further attention and explanation.

Yet, in the design and analysis of algorithms, we are forced to express ourselves unusually exactly. This implies that we need a thorough mastery and precision in understanding our language. Is it therefore surprising that its details still consume a considerable amount of time in teaching the art of programming?

The language Pascal was developed in the late 1960s in recognition of the fact that the language used is of paramount importance in programming. The authors of this book have chosen Pascal as a vehicle, and they are able to concentrate on the fundamental principles of programming and problem solving. Nevertheless, the details of the language are given due attention, and they are clearly motivated. Consequently, they appear as rules that can be understood instead of merely being memorized. The book explains the style of programming that was the guiding idea in the design of Pascal. By explicitly motivating the principles of structured programming and the corresponding features of Pascal, it provides insight instead of merely coverage.

N. Wirth

Zürich, Switzerland August 1977

PREFACE

This textbook represents the culmination of my efforts to develop an introductory programming course that would reflect the growing concern for teaching the design and development of high-quality, reliable software. It follows closely the outline of the course entitled CS1, Computer Programming I, as described in ACM Curriculum '78. (*Communications of the ACM,* March, 1979.) Too often a student's introduction to programming has been through a service course where the major concern is the syntax of some elementary programming language, and the programs are graded solely on the basis of whether or not they produce correct results. Unfortunately, these service courses often instill and reinforce bad programming habits, and later attempts to "unlearn" them are usually futile. (In this respect it is interesting to note that the students who encounter some difficulty in the course are *not* the ones without prior programming experience, but those who have already had a low-level exposure to FORTRAN or BASIC. These students are forced into rethinking their approach to programming.)

It is incorrect to assume that beginning programming students are unable to handle "higher-level" concepts. I have found it both reasonable and worthwhile to present the topics of problem specifications and algorithm development, top-down modular programming, structured coding, and program testing along with details of a particular language. When these topics are introduced during the *initial* stages of learning, they instill good programming habits immediately. The results are solutions that are well thought out, programs that are well structured, and documentation of high quality. This is simply because our students have not been taught any other way. To them, it is the normal way of doing things.

This book has three goals; in order of importance, they are:

1. Introducing *all* aspects of the programming and problem-solving process, including problem specification and organization, algorithms, coding, debugging, testing, documentation, and maintenance.
2. Introducing what constitutes *good* programming style and how to produce a high-quality finished product. These points are brought out in numerous Style Clinics throughout the text.
3. Teaching the syntax of the Pascal programming language.

I have chosen to use Pascal as the programming language because it is an excellent language for introducing these concepts. However, I did not wish merely to replace a FORTRAN-based service course with a Pascal service course. Instead, I wanted to develop a textbook that uses Pascal as a *vehicle* to introduce a range of programming concepts. Although a large portion of this book is directed, of necessity, toward our third goal, this should be viewed in its proper perspective.

Chapters 1 and 2 introduce the student to the preparatory work that must be done prior to the coding phase—problem specification and algorithm development. Chapters 3, 4, and 5 introduce the basic elements of the Pascal language. In class I treat the syntactic details to these chapters quickly and prefer to concentrate on the stylistic aspects discussed in the text and in the style clinics. Chapter 6 discusses debugging, program testing, documentation, and maintenance. Chapters 7 to 10 introduce the remaining features of Pascal, including subprograms and advanced data structures. Again, in class the syntactic rules are covered quickly, and the conceptual and stylistic details are considered at greater length. Chapter 11, one of the most important chapters in the book, discusses techniques for developing and managing large ''real-world'' problems and writing quality programs. Sufficient time should be allowed to ensure that the material in that chapter is treated adequately.

This text is currently being used in a one-quarter introductory undergraduate computer science course at the University of Minnesoty and Macalester College. The students include both computer science and noncomputer science majors. This textbook assumes no prior programming experience or extensive mathematical background on the part of the student. The programming examples have been chosen to span a wide range of numeric and nonnumeric applications. In addition, I have avoided aspects of Pascal that might be specific to a particular computer system. Where machine-dependent details are required (e.g., control cards), I have referred the student to his or her instructor for the necessary information.

I would like to thank my coauthors on the first edition, Steven Weingart and David Perlman. Although they were unable to assist in the preparation of the second edition, I have borrowed freely from their original ideas and writings.

I also thank two people whose names do not appear on the front cover but who contributed significantly to the production of the book: Rajiv Kane, who helped to test the numerous programming examples in the book, and Sandy Whelan, who typed and proofread the final manuscript. I am grateful to the many referees whose excellent suggestions were so liberally used. Special thanks must go to Andy Mickel of the University of Minnesota Computer Center who reviewed the manuscript and initiated and encouraged my interest in Pascal as a language tool for teaching programming. Without the help of these people, this project would still be only an idea.

I sincerely hope that this book will contribute to an improvement in the quality of programming instruction and to the view of computer programming as a rational and organized discipline.

G. Michael Schneider

STYLE CLINICS

CONTENTS

Chapter 1

AN INTRODUCTION TO COMPUTER PROGRAMMING

1.1 INTRODUCTION

This book is about programming. However, that statement is not as simple as it may first seem. What we mean when we use that term, and what others mean, may be quite different. What we imply by the term *computer programming* is "the entire series of steps involved in solving a problem on a computer." Too often, however, the word *programming* has been used as a synonym for the word *coding*—the process of writing statements in some existing computer language. Classes that have purported to teach computer programming have frequently been nothing more than long litanies of syntactic do's and dont's for some specific language. The worst part of this approach is that it tends to reinforce the mistaken idea that the best technique for solving problems on a computer is to take a pencil and a piece of paper, begin writing a program, and keep writing until you are done. You then hope that you have produced a valid solution. Nothing could be further from the truth. An enormous amount of preparatory work must precede the actual coding of any potential solution. This preparation involves steps such as defining exactly what is wanted, clearing up any ambiguities or uncertainties in the statement of the problem, deciding how to solve it, and roughing out the outline of the solution in some convenient notation. In fact, if this preparatory work has been done well, the coding phase, which seems the most important to many people, becomes relatively straightforward and uncreative. It becomes simply the mechanical translation of the solution for a problem into grammatically correct statements of some particular language.

In addition, just as we must spend much time and effort before we code, we still have much to do after we have finished coding. We must then grapple with the problems of detecting and correcting errors, polishing the documentation, and testing, validating, and maintaining the program.

The point we are trying to make is that computer programming is an extremely complex task made up of many individual phases, all of which are important and all of which contribute to the solution of a problem. Do not confuse the concept of programming with any single phase (e.g., coding) to the exclusion of all others. We hope this explains why the first complete Pascal[1] program does not appear until the end of Chapter 4. When constructing your own programs, the Pascal coding should be preceded by a great deal of preparatory work clarifying, organizing, structuring, and representing your solution. Failing to understand this principle is the first and greatest mistake you can make when learning computer programming.

1.2 THE STEPS INVOLVED IN COMPUTER PROGRAMMING

In the introduction to this chapter we stressed that programming involves many steps. Let us be a little more specific and describe the actual steps involved.

1. *Defining the Problem.* The inclusion of this step seems trivial. It is obvious that we must know exactly what we want to do before we can begin to do it. But this ''obvious'' phase is too often overlooked or omitted by programmers who begin their work on problems fraught with ambiguities and uncertainties. A clear understanding of exactly what is needed is absolutely necessary for creating a workable solution. The task of defining the problem will be discussed in the remaining sections of this chapter.
2. *Outlining the Solution.* Except for the simplest of problems, a program will not be composed of a single task but of many interrelated tasks. For example, a computerized payroll system would most certainly not be viewed as a single program. Instead, it will probably contain several program units that validate input data, sort and merge files, compute and print paychecks, print output reports and error logs, and keep year-to-date information. On large projects that involve a number of programs and programmers, it becomes extremely important to specify both the responsibilities of each task and how these individual tasks interrelate and interact. This is to ensure that the pieces being developed separately are designed in the context of the whole.

   Of necessity, the early programs in this textbook are short and simple and composed of single tasks. For these programs the outlining phase can probably be neglected without severe complications. However, they should be viewed as ''toys'' being used for teaching purposes only. In later chapters we will be devoting a great deal of time to program development, program structure, and the management of large, real-world programming projects composed of many separate modules.

[1]Pascal is *not* an acronym and is therefore not written out in capital letters. The language was named in honor of the French mathematician and religious fanatic Blaise Pascal (1623–1667).

3. *Selecting and Representing Algorithms*. We have now specified the various tasks and subtasks required to solve our problem. For each task we know what information we will provide and what results we want to produce. But we have not yet specified *how* the program is to accomplish its stated purpose. An *algorithm* is the specific method used to solve a problem. The algorithm may be one already developed and published in the literature or one of our own creation and design. For reasons that we will discuss later, it is not a good idea to begin immediately coding the informal specifications of an algorithm directly into some existing programming language. Instead, it is advantageous to describe the details of the proposed solution in an algorithmic representation that is independent of any computer language or machine. In Chapter 2 we will discuss algorithms and their development. In addition, we will describe in detail one specific method of representing algorithms.
4. *Coding*. Only after unambiguously defining the problem, organizing a solution, and sketching out the step-by-step details of the algorithm can we consider beginning to code.

   Your choice of which computer language to use will probably be dictated by three considerations.

   1. The nature of the problem.
   2. The programming languages available on your computer.
   3. The dictates and limitations of your particular computer installation.

   Some programming languages are general purpose; others are very specific for certain classes of problems. Some languages are very widely available; others can be run only on a very few computers. Figure 1-1 lists a few of the more common programming languages that you may encounter.

   In this textbook we will employ the language called Pascal. It is a very elegant language, complex enough to introduce important concepts in computer programming but simple enough to be a good teaching tool in a course in computer programming. Chapters 3 to 5 and 7 to 10 will describe the correct use of the Pascal language. However, our concern is not merely to teach you how to use Pascal correctly but how to use it well. We will devote a great deal of effort to developing a set of guidelines to aid you in writing ‘‘good’’ programs. These guidelines, taken together, constitute a *programming style* and will be presented both in the text and in numerous *Style Clinics* throughout the book.
5. *Debugging*. The novice programmer quickly learns that a problem is far f solved once the program has been coded and run. We must still locate and correct all the inevitable errors. This is a time-consuming and often agonizing task. Section 6.3 provides some tips and guidelines to make debugging more manageable and less painful.

Language	*Approximate Date of Introduction*	*General Application Areas*
FORTRAN	1957	Numerically oriented language. Most applicable to scientific, mathematical, and statistical problem areas. Very widely used and very widely available.
ALGOL	1960	Also a numerically oriented language but with new language features. Widely used in Europe.
COBOL	1960	The most widely used business-oriented computer language.
LISP	1961	Special-purpose language developed primarily for list processing and symbolic manipulation. Widely used in the area of artificial intelligence.
SNOBOL	1962	Special-purpose language used primarily for character string processing. This includes applications such as text editors, language processors, and bibliographic work.
BASIC	1965	A simple interactive programming language widely used to teach programming in high schools and colleges.
PL/1	1965	An extremely complex, general-purpose language designed to incorporate the numeric capabilities of FORTRAN, the business capabilities of COBOL, and many other features into a single language.
APL	1967	An operator-oriented interactive language that introduced a wide range of new mathematical operations that are built directly into the language.
Pascal	1971	A general-purpose language designed specifically to teach the concepts of computer programming and allow the efficient implementation of large programs.
Ada	1980	A new systems implementation language designed and built for the Department of Defense.

Figure 1-1. Survey of some widely used computer languages.

6. *Testing and Validation.* Getting results from a program is not enough. We must guarantee that they are the correct results. Furthermore, we must try to convince ourselves that the program will, indeed, produce correct results in

all cases, even those that have not been explicitly tested. Section 6.4 discusses the testing and validating of computer programs.

7. *Documenting*. The documentation of a program is a continual process. The program specifications from step 1, the algorithmic representation from step 3, and the program itself from step 4 can all be considered part of the documentation of a program. However, after successfully completing the program, we must ensure that our documentation is complete and in a finished, usable form. This includes both *technical documentation* for the programmers who may be working with and modifying the completed program and *user-level documentation* for the users of the program. Section 6.5 contains guidelines and standards for both levels of documentation.
8. *Program Maintenance*. As you will be discovering, this textbook is concerned with effective communication between persons, not just communication between a person and a computer. This will be evident from our treatment of topics such as program clarity, program readability, and documentation. This concern is caused by the fact that programs are not static entities. They frequently become outdated as errors are discovered, new problems need to be solved, or new equipment becomes available. Programs written weeks, months, or even years ago will frequently need to be reviewed, understood, and then modified by someone else. Unless we are careful to document what we have done and write our programs clearly, systematically, and legibly, this step can be frustrating or even impossible. Even if we always maintain our own programs we may find that time has dimmed our memories and that we require the same high-quality documentation as anyone else. Because the best possible documentation is simply a clearly written, well-organized, and well-structured program, we can in a sense say that this entire textbook is devoted to facilitating the continuing task of program maintenance.

Finally, we should stress that the programming process just described is not as linear as these eight steps may lead you to believe. Most of these steps overlap each other; for example, documentation will be written continually during program development. Many of these steps will have to be repeated; for example, as debugging uncovers errors, we will go back to recode portions of our program or to rethink the solution. The point of this discussion was simply to show that programming is a complex job. It is easy to become a good *coder;* reading this textbook and learning the rules of Pascal should accomplish that. Your goal should be a higher one: to become a good *programmer*, someone who understands and can manage the entire spectrum of programming responsibilities. However, in this fuller sense of the word, programming cannot be passively taught but must be actively learned through practice and experience. When accomplished, however, it is a much more creative and rewarding experience.