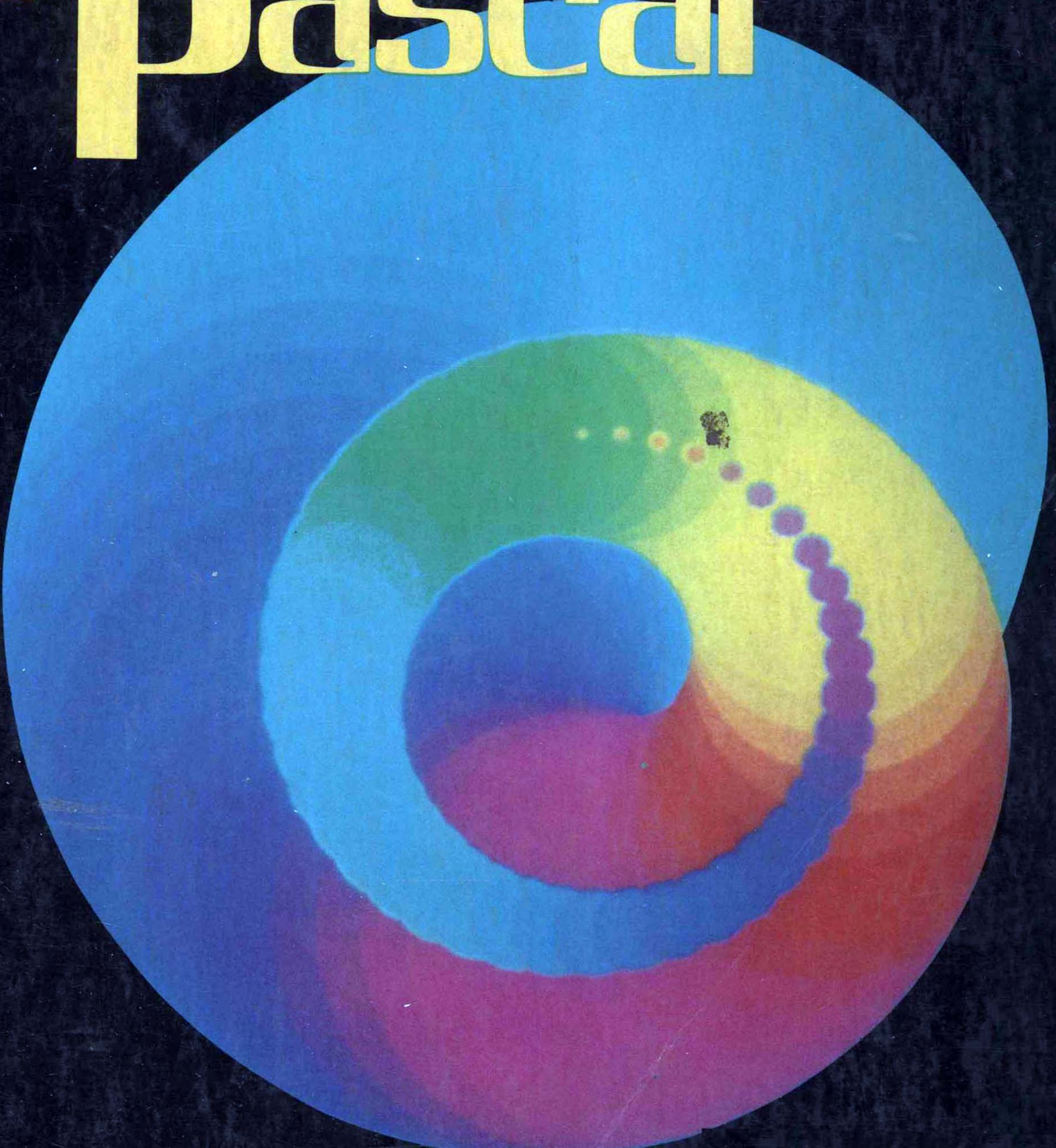
SEYMOUR V. POLLACK

introducing pascal



Introducing Pascal

Seymour V. Pollack

Department of Computer S
Washington University
St. Louis, Missouri

Holt, Rinehart and Winston

New York Chicago San Francisco Philadelphia
Montreal Toronto London Sydney Tokyo
Mexico City Rio de Janeiro Madrid

Library of Congress Cataloging in Publication Data

Pollack, Seymour V.
Introducing Pascal.

Includes index.
1. PASCAL (Computer program language) I. Title.
QA76.73.P2P64 1983 001.64'24 83-10756

ISBN 0-03-060563-6

Address correspondence to:
383 Madison Avenue
New York, New York 10017
All rights reserved
Printed in the United States of America
Published simultaneously in Canada

3 4 5 039 9 8 7 6 5 4 3 2 1

CBS COLLEGE PUBLISHING
Holt, Rinehart & Winston
The Dryden Press
Saunders College Publishing

Preface

Although Pascal made its debut in the midst of hundreds of existing programming languages, it is revolutionary in that it is the first major programming language based on a perception of programming as a systematic process. While earlier languages include features that offer varying opportunities for structured programming, the presence or absence of such features is more a matter of circumstance than design. Most of these languages were conceived and implemented prior to the emergence and crystallization of structured programming concepts. Subsequent extensions to these languages have been constrained by the necessity to remain compatible with their original syntactic rules.

Pascal, on the other hand, has no such ancestral restrictions. It reflects an explicit concern with structured programming. In fact, a major design objective is to encourage the use of constructions and practices that are associated with the production of clear, reliable programs. This is done to a large extent by incorporating language features that encourage structured programming and erecting barriers that make it awkward or difficult to avoid structured programming practices.

Given this orientation, it is useful to view Pascal as the beginning of a new generation of programming languages. This book reflects that view in that it accepts structured programming as a rational approach to program development and it presents Pascal as a reasonable vehicle for supporting this approach. There is no need at this stage of the discipline to defend structured programming and argue against the ad hoc approach that preceded it.

Pascal's organizational simplicity, when combined with the educational advantages inherent in Pascal's design, results in a powerful vehicle for developing good programming habits and acquiring the necessary programming skills relatively painlessly. Thus the student is learning to write programs in a clear and orderly way while building familiarity with algorithmic processes.

This book is designed for students who have not had previous experience with Pascal (or any other programming language for that matter). Its extensive use for example makes it equally suitable for a class or for self-study. If the student has worked with another programming language (on any system), it may be appropriate to bypass the first chapter.

There is no intent to present Pascal in its entirety. Although Dr. Nikalus Wirth, its developer, envisioned a standardized language that would be supported intact on a wide variety of computing systems, this is not quite the way events developed. Instead, various organizations, each seeking to exploit the language in some particular way, have used Wirth's standard version as a nucleus which they have embellished with an arbitrary assortment of additional features. (Some of these are seen as absolute necessities that cure basic deficiencies or oversights; others are viewed as frippery that adds unwanted complications to a language designed to be simple. Ultimately, each individual must judge the situation with respect to his or her interests.) Fortunately, these various excursions, despite their diversity, have been built around the original (standard) version so that its features form a subset of most (if not all) of Pascal's dialects.

Thus, rather than trying to accommodate the morass that such uncontrolled growth inevitably produces, this book uses standard Pascal as a vehicle for presenting well-illustrated discussions of sound programming principles and practices. (Some specialized features have been omitted and an occasional extension, explicitly earmarked as such, has been included to amplify a particular point. However, the book's flavor and direction center around standard Pascal and the principles that it embodies. The foundation thus provided makes it relatively simple for the programmer to use any additional features once he or she becomes acquainted with their operational properties.

Another feature of the text is worth noting here. Pascal is neither an "Interactive" language nor a "batch-oriented" language. Since it is intended to be a general-purpose language with particular advantages for teaching sound programming principles and techniques, it (rightly) avoids such issues. Instead, the selection of an operating mode is treated conceptually as an external process that depends on the requirements and characteristics of the particular application. As far as Pascal is concerned there are facilities for implementing a program either way. Consequently, this book pays attention to both types of implementations so that the student is not nudged toward a prejudice in either direction. Each mode has its uses, and the student is motivated to learn techniques for both so that he or she can select the proper one on a rational basis.

I would like to thank Brete Harrison and Paul Becker of CBS College Publishing and Rachel Hockett of Cobb and Dunlop for their valuable suggestions and help. Finally, my continued gratitude and love go to Sydell, Mark, and Sherie Pollack. These are truly wonderful people.

Seymour V. Pollack

Contents

Chapter One

Introduction

In this book we shall be concerned with the features of the Pascal programming language and the techniques for using them to develop effective problem solutions on a computer. Unlike the names of most other languages, Pascal is not an abbreviation for anything. The language is named in honor of Blaise Pascal (Figure 1.1), a brilliant French mathematician of the seventeenth century. He is generally credited with building the first successful mechanical adding machine. Although there is no direct historical connection, Pascal's device (Figure 1.2) is considered the starting point in a long line of arithmetic machines preceding today's electronic digital computer.

The Pascal programming language, developed by Dr. Niklaus Wirth, was introduced in 1971. Although there already were several hundred programming languages in existence at the time, it would be misleading to think of Pascal as just another one to add to the list. Pascal's importance lies in the fact that it explicitly seeks to support a view of programming as a systematic, orderly activity. While the earlier major programming languages do not promote chaos (some people will give you an argument about that), they represent ideas and attitudes about programming that date back to the early 1950s. Accordingly, these languages place primary emphasis on making it convenient to specify certain kinds of computations considered to be useful. With few exceptions, these languages pay only incidental attention to the organization of these computations and to the data on which they are performed.

Over the years, the programming process has been the subject of extensive study. As a result, we now appreciate the importance of treating a program as a manufactured product. While it is not a physical item (we do not think of a program as something that can be carried around in a bucket), it shares many attributes with a physical product: It consists of several parts (called *modules*), each of which is designed for a particular purpose. Moreover, these modules must work with each other in certain predefined ways in order for the overall assembly to function properly and effectively as a unit. This similarity to a manufactured physical item compels us to

Figure 1.1 Blaise Pascal (Courtesy of Science Museum, London)

recognize the fact that the programming process lends itself to the same orderly, systematic approach underlying the design and development of any other product. Structured design and structured programming, both of which are still evolving, stem from this recognition.

Having said all of that, we can now establish Pascal's significance: It is the first major programming language in which program design and organization are central concerns. Therefore, its primary features are arranged so that they encourage good structure by making it easy to specify. Pascal goes even further by imposing requirements that make it awkward or difficult to avoid good structure. The result is a powerful tool whose mastery can be of great help in using the computer to solve problems effectively.

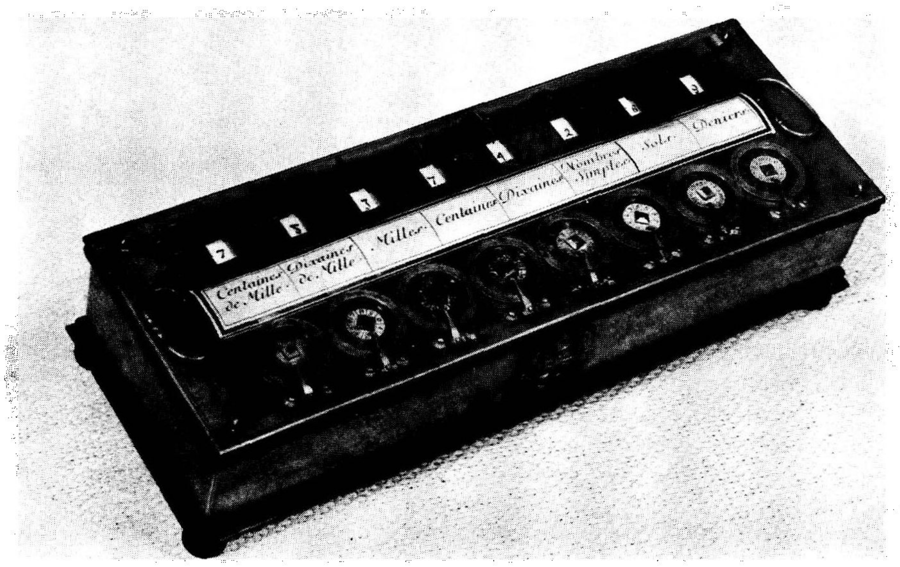


Figure 1.2
Mechanical Calculator (Courtesy of Smithsonian Institution)

1.1 SOLVING PROBLEMS SYSTEMATICALLY

The computer does not know all the answers. Strictly speaking, it does not "know" anything. Its circuits are designed to move information around in certain ways so that particular elementary operations can be performed on that information. However, it is up to us to combine these operations so that the overall effect is to produce useful results. Thus, when we use a computer to solve a problem, we are the ones who determine how the problem is to be solved, and the computer carries out the solution by following our directions. This relationship between the computer and the human programmer is an important one: It means that the programmer knows how a certain problem is going to be solved before he or she calls on the computer to help out. The method of solution, called an *algorithm*, is worked out beforehand and is expressed as a series of steps, each of which brings us closer to the final result. We bring the computer into the picture by restating the algorithm as a *program*. This is the same series of steps, described in the precise vocabulary of a programming language.

Before we can write a program, it is necessary to develop an algorithm and describe it so clearly that its conversion to a correct program is relatively uncomplicated. This is when the problem solving takes place, not when the program is written. Although there is no exact, foolproof recipe for solving problems, there are orderly steps that help increase the likelihood of a successful solution. We shall outline these steps briefly in the next few sections.

1.1.1 Identifying the Problem

It is difficult to imagine the amount of time, money, and effort spent on developing extensive, ingenious solutions to the wrong problems. Common sense would indicate that this can be avoided by making sure the problem is clearly defined before one plunges ahead and starts solving. Yet, these situations continue to arise. Recently, the people running a hospital in a certain city observed that their patients were staying one or two days longer than those admitted to other hospitals for the same types of operations or treatments. The problem was spotted immediately (or so they thought): Obviously, the physicians kept the patients too long after surgery. The hospital launched an extensive and vigorous campaign aimed at encouraging the doctors to get their patients up and moving as soon as good practice allowed. The physicians insisted they were doing that anyway, but promised to pay particular attention to this issue. Nothing changed. Finally, the real problem was discovered: A patient could not be discharged until the bill was ready, and it was taking the accounting department a day or two longer to prepare the bill than it did at other hospitals.

Because of situations like this, many organizations have made it a rule to develop and agree upon a precise, written statement of the problem to be solved before any work is started on a solution.

1.1.2 Finding a Suitable Solution

It is easy to say, "Now that we know exactly what the problem is, the next step is to find an effective way to solve it." Sure. Nothing to it. Of course, there is no step-by-step procedure that leads us from problem to solution. In fact, we cannot be sure that there is a solution just because we know what the problem is. Because of these uncertainties, this step is usually the most difficult one in the entire process. It is the one where much of the creative effort and ingenuity is concentrated. Entire libraries can be filled with writings seeking to learn the nature of this process. Several are given as references at the end of this chapter.

For our purposes, the solution to a problem takes the form of an *algorithm*. We can define an algorithm as a sequence of steps or rules that meets certain requirements:

1. An algorithm must be *finite*. That is, it must come to a stop sooner or later. For instance, suppose we were faced with the problem of producing a particular shade of blue paint. Our proposed solution is as follows:

```
Get a can of base paint and a can of blue coloring.
Open both cans.
Add blue coloring to the base paint until the color is right.
```

While this procedure seems harmless enough, it might not be finite. For example, there could be a case where the base color is too blue to begin with. According to the instructions, this is not the right color. But the procedure says to keep adding till the color is right. So, in goes more and more blue, indefinitely. This is not as ridiculous as it sounds. Remember that computers cannot exercise judgment. It is up to us to provide a specific mechanism that will guarantee a stopping point regardless of the conditions under which the algorithm operates. For this little procedure, we might specify the following revision:

```
Get a can each of base color, white, and blue color.
Open the three paint cans.
IF the base color is too blue,
THEN
   Add white paint, a little at a time, until the color is right.
ELSE
   Add blue paint, a little at a time, until the color is right.
```

We may be able to think of other situations for which this procedure will not be finite, but we shall not belabor the issue. The basic point is that there must be a way to bring our procedure to a successful conclusion.

2. An algorithm should be *precise*. This means that each of the steps must be described in such a way that it can be performed by whoever (or *whatever*) will be carrying out the process. In our little example, which lacks precision, we would have to specify the amount of blue or white paint to add (how much is "a little?"), and we would have to describe how to determine when the right color is produced. For a skilled colorist, this might be as simple as saying, "When the mixture matches this color sample, you have the right color." For someone who is less sensitive to color, the directions might be quite different: "When the mixture gives a reading between 407.5 and 423.3 on the Schmugelsky Coloramic Pigmentovacutron, you have the right color." When the algorithm is to be transformed into a computer program, the requirement for precision means that each step must be expressed in such a way that it can be carried out on a computer.

3. An algorithm should be *general*. This means that the procedure should not be so limited that it solves only one specific problem for one specific case. Instead, it should be capable of producing satisfactory solutions for a variety of cases. For example, our little color mixing procedure does what we need it to do as long as we want that particular shade of blue. We could generalize it by expanding its capabilities to include other shades of blue, or even other shades of other colors.

There are many instances where we can come up with several different solutions for a problem. For instance, suppose we are driving along and suddenly the ride becomes much bumpier without any noticeable change in the road. Moreover, steering becomes more difficult. After stopping the car and inspecting it, we find that one of the tires is flat. Before discussing possible solutions, we must agree on what the problem is. In this instance there is little difficulty on that score: Our journey has been interrupted, and we wish to resume it as soon as we can. When we state the problem that way, we can think of numerous potential solutions. Here are just a few:

1. Abandon the car and continue on foot.
2. Buy another car and continue in that one.
3. Reinflate the tire and continue the trip.
4. Get in touch with a service station and have one of their mechanics replace the tire.
5. Flag down a passing motorist and persuade that party to transport us to our destination.
6. Flag down a passing motorist and persuade that party to replace the tire.
7. Call home and get a loved one to come out and replace the tire.
8. Replace the tire ourselves and continue on our way.

Sometimes we can narrow down the choices by rejecting some as being clearly unsuitable. In our example, solutions 2 and 3 are in that category. We probably would not need much convincing to reject solutions 5 and 6 as well. After that, the choice of the most "appropriate" solution may become more complicated because what is "appropriate" will depend on the circumstances. For example, solution 1 may be best after all if we are close to our destination and we must get there promptly. Thus, in many situations, selection of an effective solution method can be the most challenging part of an information-processing project.

1.1.3 Decomposition of a Problem

Once we identify a problem, it is often impossible to solve the entire problem all at once. More likely than not, the problem is too complicated for us to be able to keep track of all the possible twists and angles at the same time. When this happens, it is useful to break up the problem into a collection of interrelated subproblems. Each of these is small enough so that we can handle it comfortably, as a complete entity. This process of *decomposition* makes it easier to develop a solution for each little problem. As a result we can produce a complete, precise definition of what each piece is supposed to do and how it fits with the connecting pieces. The stage is set, then, for each of these components to be developed separately.

To illustrate, let us return to our disabled automobile. Suppose we decided that the best solution under the circumstances was to replace the flat tire with the spare tire. When we look at the solution more closely, we can identify several steps, each of which embodies a little problem of its own:

1. Remove the spare wheel/tire and the necessary tools from the trunk.
2. Remove the wheel on which the flat tire is mounted.
3. Install the wheel on which the spare tire is mounted.
4. Place the wheel removed in step 2 in the trunk, along with the tools.

The activities required to carry out a given step become increasingly clear as we focus even more closely. For example, let us consider step 2 in more detail. The actual removal of the wheel cannot occur without some preparation.

2. Remove the wheel on which the flat tire is mounted:
- **2.1** Remove the hubcap and retaining nuts from the wounded wheel.
- **2.2** Find an appropriate place for positioning the jack.
- **2.3** Set up the jack.
- **2.4** Raise the wheel by means of the jack until the wheel no longer touches the ground.
- **2.5** Remove the wheel.

If necessary, decomposition continues by dissecting each of these subproblems in turn, until each activity has been reduced to a series of simple steps that are completely understood.

The same process applies to a problem solution involving a computer. Eventually, the result will be a program, but we know nothing about that program's details at this point. Rather, the decomposition process tells us how that program divides into pieces and what each of these pieces needs to do.

1.1.4 Identification of Component Details

At this stage of the game, we do not yet know exactly how each piece identified during the decomposition will do its job. In fact, we may not know whether each of these little problems *has* a solution or not. The whole point of the decomposition process is to make it easier to deal with these issues one at a time. Once the pieces are known, we can focus in on each one to determine its characteristics in detail. As a result, each piece will be completely defined: We shall know what it does, what it needs to do its job, and precisely how that job is done.

The idea behind this step can be illustrated with a simple example:

Suppose our problem is to produce a device for playing phonograph records. As a result of the decomposition step, we identify the need for a number of components, one of which is a turntable on which the record is to spin. (Studies during the solution-finding stage have convinced us that it will be better to have a spinning record and a passive tone arm than to use a design in which the record stands still and the needle races around the grooves.) We know upon entering this stage that the turntable must be able to hold records of a certain size, and that it must be driven at a certain speed. Now, during this stage, we can deal with such questions as the turntable's thickness, the material out of which it is to be manufactured, the distribution of its weight throughout its surface area, and other such concerns. By the time this stage is completed, we shall have developed enough information about the turntable to enable it to be built. The same will be true for the other parts. In the case of a computerized problem solution, then, *this stage produces complete specifications for each part of what* eventually will be a computer program. These parts are called *program modules* or, simply, *modules*, and we shall use the term in that context. Note that we haven't written any programs yet. That comes in the next step.

1.1.5 Coding

It is at this point that we are ready to express our solution as a sequence of program statements so that a computer can be used to carry it out. If we did a reasonable job during the previous stages, this part of the process will be like a translation from one "language" to another. Although in this book we shall be translating to Pascal, it is important to point out that the description of each module should contain enough detail so that it can be restated in any one of a variety of programming languages. As a matter of fact, when the modules' details are being worked out, the particular programming language to be used does not enter the picture and should not influence that work.

The process of writing such program modules is called *coding*. It is important to understand that we are not developing a solution method at this stage. That has been done already in previous stages. We know what we want to do, and we know how we want to do it. We also know what we want to tell the computer. When we write the code, we convert the description of our intentions from a form that the computer cannot use to a form with which it can work.

Pascal itself is organized to make the coding process particularly convenient, especially when it is driven by an orderly set of specifications. We shall take full advantage of this convenience by emphasizing the preparation of clear, unambiguous descriptions for program modules. The pertinent discussion begins in Section 1.3.