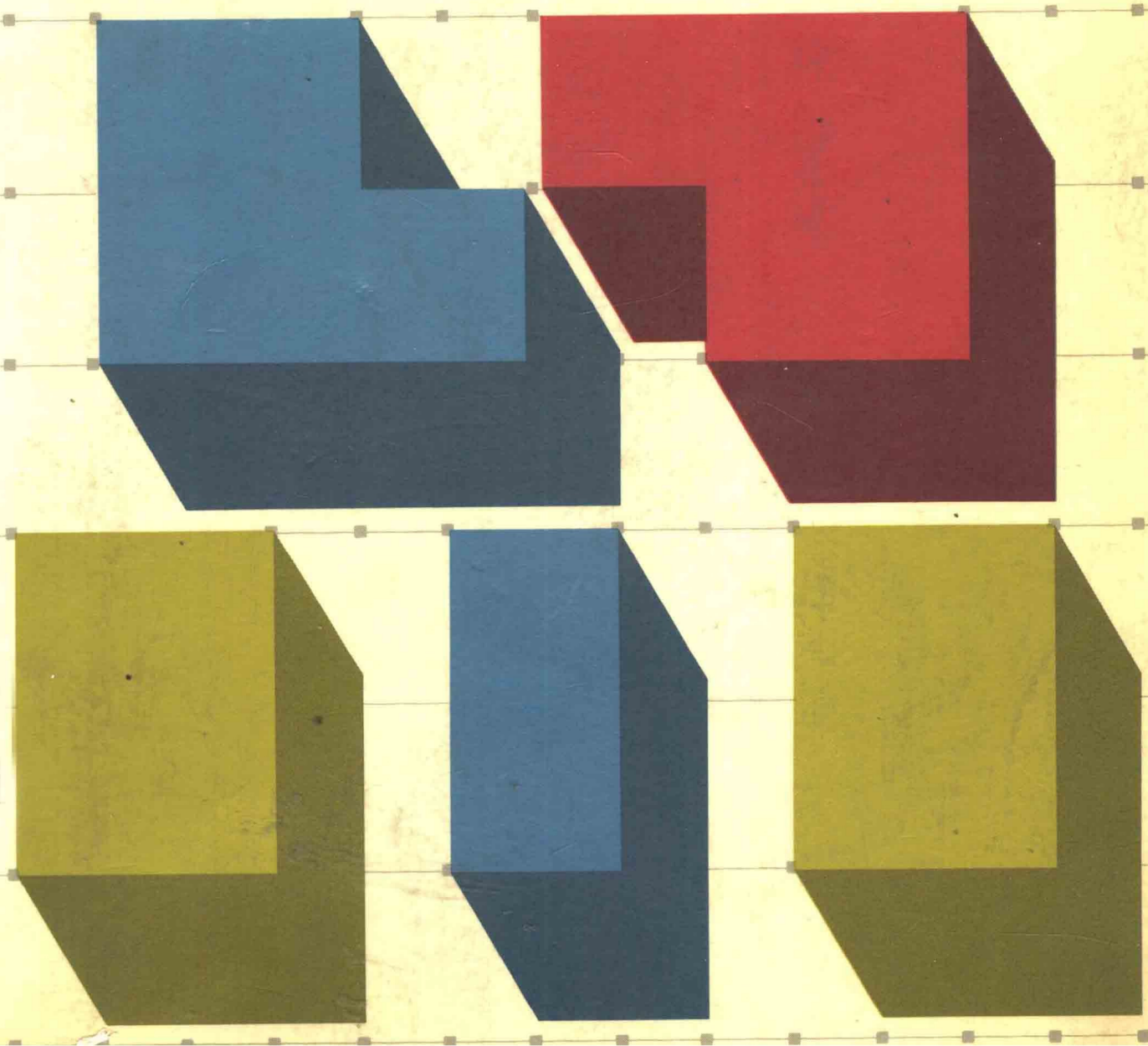
Program Design and Data Structures in Pascal

Charles W. Reynolds



Program Design and Data Structures in Pascal

Charles W. Reynolds
James Madison University

Wadsworth Publishing Company
Belmont, California

A Division of Wadsworth, Inc.

Computer Science Editor: Frank Ruggirello
Production Editor: Sandra Craig
Managing Designer: MaryEllen Podgorski
Print Buyer: Ruth Cole
Text and Cover Designer: Janet Bollow
Copy Editor: Brenda Griffing
Technical Illustrator: Barbara Barnett
Compositor: Graphic Typesetting Service
Signing Representative: Bob Podstepny

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10—90 89 88 87 86

ISBN 0-534-06294-6

Library of Congress Cataloging-in-Publication Data

Reynolds, Charles W., 1948–
Program design and data structures in Pascal.

Bibliography:
Includes index.
1. PASCAL (Computer program language) I. Title.
QA76.73.P2R49 1986 005.13'3 85–24965
ISBN 0-534-06294-6

In some disciplines, the introductory courses survey fundamental concepts and introduce subordinate disciplines. In other disciplines, the introductory courses teach fundamental skills essential for understanding the discipline. Computer science is one of the latter. Although computer science is not the study of computer programming, you cannot study computer science until you can understand moderately large programs.

With this in mind, the first year of an undergraduate computer science education should be devoted to learning how to write programs in a high-level language, as the Curriculum 78 Model of the Association for Computing Machinery suggests. This first year is divided into two courses, CS1 and CS2. *Program Design and Data Structures in Pascal* is intended for use in the second course, CS2, and its goal is to take students from being comfortable with programs of 50 to 100 lines and three or four procedures to being comfortable with programs of 500 lines and thirty or forty procedures.

Top-Down Design and Abstract Data Types

Novices have a strong desire to understand "what is really happening in a computer," yet computers are so complex that no one, no matter how experienced, can do this. Humans understand computers as layers of abstraction. We then focus our attention on one layer at a time. We understand how one abstract layer uses the layer below it, and we understand the services that layer provides to the layer above it. We then move our attention up and down this hierarchy, understanding each layer in isolation. The only way to learn this use of abstraction is by experience with it, and *Program Design and Data Structures in Pascal* provides that experience.

Through numerous programming projects described using top-down design, this book helps students learn to use layers of abstraction. Using top-down design, the initial understanding of a problem is broken down into smaller problems with structured programming. Solving the smaller problems in the same fashion leads to a solution of the original problem. Another key element in our design methodology is the use of bottom-up design with abstract data types. An abstract data type is a widely useful group of procedures and functions that collectively manipulate and examine some conceptual data object. In the portions of a program that use the procedures and functions of an abstract data type, we are not concerned with how the procedures and functions are implemented, only with what can be done with them. In the portions of a program that implement the procedures and functions of an abstract data type, we are not concerned with how they are used, only with how to implement them.

Prevention of Logic Bugs

Professional programmers do not debug programs: they write programs that do not have bugs. To back off from this extreme position somewhat: professional programmers do encounter bugs in their code, but these bugs are typically trivial, easily identified, and easily corrected. To produce a correct

program in five compilations is acceptable. To use fifty compilations is not acceptable. The difficult bugs are the ones that occur at the design level, the so-called logic bugs. These are the bugs that give students the most trouble, and these are the bugs that good programmers don't introduce into their code. This book will help students avoid these bugs because each design is described in detail. If students read and understand the designs, they will have no logic bugs. If, by the end of this textbook, students feel that producing a good design is challenging and that coding a fully described design in a programming language is trivial, then they will have learned what this book has to teach.

Programming Projects

Every chapter contains programming projects. The design of the project in Chapter 1 is completely described and then implemented so that students can see what will be expected in later chapters. Starting in Chapter 2, every chapter contains one or two major programming projects whose designs are carefully described for students. Beginning in Chapter 4, there are also programming projects that are not designed so students can exercise the design skills they are developing through the course of the book. I usually give students a week to do one of the predesigned projects and two weeks for projects they must design. Thus, there is not enough time to debug a program unless it is fundamentally correct when first written. This is crucial in a computer science education; the ability to write large quantities of source code correctly on the first try is not what computer science is, but it is a prerequisite to being able to do computer science on a professional level.

Organization

The complementary techniques of top-down design and bottom-up design with abstract data types are used in programming projects through the book, but Chapter 1 is concerned exclusively with these issues and introduces them in the context of an interesting robot simulation. Then several common abstract data types are introduced in Chapters 2 through 5. These include input files, stacks, queues, tables, and character strings. When first introduced, each is implemented as simply as possible using a static sequential memory allocation. These chapters emphasize the notion of an abstract data type, its abstract specification, and the integrity of its interface. The exercises in these chapters stress these topics, and the projects illustrate typical uses of these common abstract data types in top-down designs.

In addition to introducing the table abstract data type, Chapter 4 introduces the importance of time complexity by studying the binary search algorithm. Besides introducing the character string abstract data type, Chapter 5 introduces the importance of space complexity by studying the allocation of variable-length strings in a global character heap.

In Chapters 6 through 10, the focus shifts to the importance of time and space complexity. In Chapter 6, recursive programming is introduced, and then, as an application, the Quicksort algorithm is studied in depth. Chapter 7 introduces linearly linked allocations by providing an alternative implementation of the stack and queue abstract data types first studied in Chapter 3. Chapter 8 continues this study with a linearly linked allocation of the table abstract data type first studied in Chapter 4. Both Chapters 7 and 8 stress the importance of linearly linked allocations as the flexibility of their space utilization. Chapters 9 and 10 study hashing algorithms and the binary search trees, respectively. The algorithms are used as the basis for a third and fourth implementation of the table abstract data type. The emphasis in both chapters is on the speed with which an arbitrary table item can be found.

Chapters 5 through 10 thus introduce data structures and the time and space trade-offs that must be considered in choosing the implementation of an abstract data type. The exercises in these chapters are mostly concerned with manipulations of the data structures being studied. The projects in these chapters are more difficult than those earlier in the book, requiring the use of several abstract data types in each.

For quick reference, the abstract data type specifications are drawn together in an appendix that also includes a page index to the implementations of each abstract data type. A composite glossary/index provides definitions for all major terms and also includes descriptions of all procedures and functions arranged alphabetically with page references.

Acknowledgments

I would like to thank the reviewers of this book: Nancy Dickerson, Black Hawk College; John Donaldson, University of Akron; Ken Friedenbach, University of Santa Clara; Judy Gersting, Indiana University–Purdue University at Indianapolis; Henry Gordon, Kutztown University of Pennsylvania; Nancy Griffeth, Georgia Institute of Technology; Samuel Gulden, Lehigh University; Greg Jones, Utah State University; Ernst Leiss, University of Houston, Central Campus, Houston; Lorraine Parker, Virginia Commonwealth University; and Douglas Re, Skyline College.

Charles W. Reynolds

Contents

Computers are capable of processing immense detail at immense speed with immense accuracy. At least, this is the way they appear to us. That they produce information at such incredible speed, while making no mistakes in their calculations, only heightens our amazement. We on the other hand are lazy; we get bored; we avoid mindless, repetitive tasks; and we are extremely error prone.

The computer does do very well what we hate to do, and in fact what we do very poorly. But, we must also appreciate ourselves, for we do well what the computer is unable to do for itself. Only we can understand; only we can see meaning in all the details; only we can write the programs. It is truly a symbiosis: the computer is utterly incapable of writing programs or of even deciding what programs should be written, and we would be bored or often incapable of performing for ourselves the calculations we program for the computer.

But how exactly, with our intelligence and our ability to understand, do we see meaning in the bit manipulations of a computer? The answer is of course that we don't look at the bits and we don't look at the manipulations of them. Rather, we group the bits into larger groups and call them numbers and characters and instruction codes. We then group these things into yet larger groups and call them vectors, character strings, tables, I/O buffers, and code segments. We also group bit manipulations and call them read operations, arithmetic expression evaluations, search algorithms, and all sorts of other meaningful names.

This process of grouping things into single concepts and then grouping concepts into ever higher levels of concepts is called abstraction.

Here's an experiment. Try memorizing the following list of nouns. Then, without looking, write the nouns down, in any order. Don't just go on reading; do the experiment.

BROCCOLI PENCIL ERASER CAULIFLOWER
CHAIR STAPLER TABLE CABBAGE SOFA

If you were able to memorize the list, ask yourself how you did it. Didn't you do so by using the groupings:

BROCCOLI CAULIFLOWER CABBAGE
PENCIL ERASER STAPLER
CHAIR TABLE SOFA

If you were unable to memorize this list, wasn't it because you wouldn't expend the effort required to see the groupings? That effort that you were or were not willing to expend is the most significant power of your mind. It is the power by which your mind imposes on a group of nouns a unity that is not readily apparent in the presentation. It is the power by which your mind imposes structure on what is otherwise unstructured. This kind of thinking is built into your brain. It is called abstraction.

The ability to think in abstractions is what makes us human. It is our defining trait, and it is the basis for our intelligence. It is important to keep this in mind, since it is essential to the human–computer symbiosis. The computer's speed and accuracy and the human's capacity for abstraction form a powerful symbiotic relationship only if the role of each is understood and appreciated.

Abstraction by Procedures [1.1]

Programming language designers early recognized our tendency for operation abstraction and supported it with the subprogram, something variously called subroutine, procedure, or function.

For the moment, let's discuss only the Pascal procedure. Our comments will apply equally well to all forms of subprogram.

As generally introduced in the beginning texts on programming, the advantage to the use of procedures is their elimination of repetitive code. Suppose we want to draw the figure

```
---------------
:             :
---------------
          >>>>>>>>>>>>>>>
---------------
:             :
---------------
```

Of course we can use Pascal output statements and the characters -, :, and > to generate the figure, as follows:

```
PROGRAM DRAW;
BEGIN
      WRITELN('---------------          ');
      WRITELN(':             :          ');
      WRITELN('---------------          ');
      WRITELN('          >>>>>>>>>>>>>>>');
      WRITELN('---------------          ');
      WRITELN(':             :          ');
      WRITELN('---------------          ');
END.
```

This program obviously produces the desired figure, but what is that figure? Realizing that the first three write statements are identical with the last three, the programmer also could have used a procedure:

```
PROGRAM DRAW;
  PROCEDURE DRAWTREAD;
  BEGIN
     WRITELN('---------------           ');
     WRITELN(':              :           ');
     WRITELN('---------------           ');
  END;
BEGIN
   DRAWTREAD;
   WRITELN('            >>>>>>>>>>>>>>>>');
   DRAWTREAD;
END.
```

The argument is that we have thus eliminated the repetitive code. We have used the three write statements once in the procedure DRAWTREAD. Then we use that procedure twice to draw two treads, one above and one below the center line of >>>>>>>>>>>>>>>>.

But this is not the most important advantage to the use of procedures. Rather, it is their support for the process of abstraction. In the last version, the programmer has indeed eliminated the repeated use of three write statements, but, much more important, those three write statements have a name: DRAWTREAD. This name is immensely helpful in understanding the program. It names a single concept, drawing a tread, that gives meaning to what otherwise appear to be three meaningless write statements. We might now begin to wonder what the programmer has in mind when drawing this figure. Suppose the programmer goes a step further and writes two other procedures:

```
PROGRAM DRAW;
  PROCEDURE DRAWTREAD;
  BEGIN
     WRITELN('---------------           ');
     WRITELN(':              :           ');
     WRITELN('---------------           ');
  END;

  PROCEDURE DRAWGUNTURRET;
  BEGIN
     WRITELN('           >>>>>>>>>>>>>>>>');
  END;

  PROCEDURE DRAWTANK;
  BEGIN
     DRAWTREAD;
     DRAWGUNTURRET;
     DRAWTREAD;
  END;

BEGIN DRAWTANK END.
```

Here, the programmer has not eliminated any repetitive code. But now a

meaningful name has been given to the acts of drawing the center line and drawing the entire figure. If there was previously any doubt that the entire figure is an armored tank, as known and loved by video game enthusiasts, that doubt has been erased. Suddenly, we no longer see the figure as a combination of -, :, and >. We see the parts of a tank—the gun turret and the two tractor treads.

As the example illustrates, procedures are used to make groups of instructions meaningful by giving them a name.

If the instructions are carefully chosen and if the name applied to them is accurate, then the procedure performs a single meaningful task that we think of as one operation. We then forget how the procedure is performed, remembering only what it does. This forgetting takes no effort on our part; it is natural to our intelligence. This is, in fact, what abstraction is. It is a conceptual wall around the procedure, a wall that is opaque in both directions because we forget what is on the other side. From inside, we need know only what the procedure does to be able to understand how it does it; we don't need to remember the uses to which the procedure is put. From the outside also, we need to know only what the procedure does to be able to make use of it; we don't need to remember how the procedure performs its task.

The use of procedures helps us understand programs only if they are chosen to implement concepts that are easy to understand. Then we can remember the concept and forget the details of the procedure implementation. But if the procedures are not well chosen, our ability to remember by abstraction will fail us and we will not be able to understand the program.

Essentially, useful procedures implement concepts that can be understood in isolation from the program in which they are used.

Four criteria are indicative of good procedures:

1. Does the procedure have a descriptive name?
2. Is the procedure easily testable?
3. Is the procedure easily modified?
4. Is the procedure potentially reusable?

The first criterion asks whether the procedure name accurately reflects what the procedure does. If no such name can be found, then the procedure is not well conceived—it does not implement a simple abstract concept.

By the second criterion, we determine whether the procedure can be easily tested in isolation from the rest of the program. This is important in testing large programs because if isolated testing is not possible, it may be difficult to determine the cause of an error. If a procedure implements an independent, understandable concept, then it can be tested in isolation.

The third criterion of a well-chosen procedure asks whether changes in the procedure require changes in other portions of the program. Programs

are changed frequently during the years of their use. This is difficult to accommodate if the procedures that make up a program cannot be modified in isolation from other procedures of the same program. If the procedure implements an isolated concept, then it is easy to modify.

Finally, we check to see whether the procedure is potentially usable from several different points in a program or in other programs. Again, this is a good indication that the procedure implements an isolated, independent idea that we will be able to hold in our minds as a single concept.

Top-Down Structured Programming [1.2]

So the use of procedures is good because it supports the mode of intelligence that we bring to the problem of understanding programs. We have also said that good procedures implement self-contained, independent concepts. It is some help to know what we want. But we are still a long way from knowing how to write good procedures. How are we, in a practical sense, to decide what the procedures should be and what their names, parameters, and global variables should be?

The issue of how one practically chooses procedures is called **program design**. There are two techniques that help with this question. They are called top-down design and structured programming.

Top-down design is a technique for program design that tells us to begin a design by first understanding the problem. Decompose this understanding into smaller problems such that, if we could solve the smaller problems, then we could solve the original problem.

The original problem is thus decomposed into a number of smaller problems, each of which is solved in the same way. Iterate the process until the problems left unsolved can be solved directly by program statements. Top-down design requires some courage—courage gained mostly from experience with it.

Structured programming is a technique for performing the problem decomposition required by a top-down design. It requires that we decompose a problem in one of three ways:

1. Sequencing
2. Iteration
3. Selection

The best way to discuss these three methods of structured decomposition is in the context of an example.

sented as rectangles. The robot, in moving to the pot of gold, must not walk into any buildings; it must go around.

In Figure 1.2, the pot contains only 4 pieces of gold, the robot is initially facing west, and there are a half-dozen buildings to be avoided. To keep the geometry simple and the intelligence required of the robot minimal, we will assume that all rectangles are aligned so that their edges are horizontal and vertical. We will also assume that the rectangular buildings are far enough apart to allow the robot to move between them. This means that there is at least one line of dots between any two buildings. We want a program that will print instructions that move the robot from its initial position to the pot of gold without hitting any buildings and will then indicate how many gold pieces to pick up.

Solving the Problem with Top-Down Structured Programming

Top-down design tells us to understand the original problem by decomposing it into simpler subproblems. Structured programming tells us to decompose a problem in one of three ways: sequencing, iteration, and selection.

Sequencing is used when a problem can be decomposed into parts such that, *if the solutions to the parts are executed one after another, the original* problem is solved.

Pascal supports this mode of decomposition by allowing any number of instructions to be enclosed in a `BEGIN..END` pair and separated from one another by semicolons. In the robot/gold problem, we must do three things in sequence: initialize the map in the program memory, print instructions that move the robot to the pot of gold as indicated on the map, and print instructions that indicate how many pieces of gold are to be picked up. So we decompose to three procedures `MAKEMAP`, `GOTOGOLD`, and `PICKGOLD` that do exactly these three things. Then we can write `GETGOLD` by sequencing:

```
PROCEDURE GETGOLD;
BEGIN
      MAKEMAP;
      GOTOGOLD;
      PICKGOLD;
END;
```

This is sequencing. We began with the problem of printing instructions for retrieving gold. We decomposed that problem into three subproblems that, when solved, permit a solution to the original problem. This is top-down design. We don't yet know how to implement the procedures `MAKEMAP`, `GOTOGOLD`, and `PICKGOLD`. But we have broken the problem down somewhat.

level abstract point of view. The robot must make repeated moves toward the pot of gold. Decompose to a procedure GOCLOSER that prints instructions that move the robot closer to the pot of gold. We don't know how this will be done, but we do know that if we can move closer to the pot of gold, we can repeat this motion until we are actually there. So, in addition to the procedure GOCLOSER, we need to be able to test whether the robot has arrived at the pot of gold. This is a further decomposition to a boolean function ATGOLD, which returns a TRUE or FALSE value depending on whether the robot is or is not at the pot of gold. Again, we don't know how this will be done. But we can now write the procedure GOTOGOLD by iteration:

```
PROCEDURE GOTOGOLD;
BEGIN
      WHILE NOT ATGOLD
      DO GOCLOSER
END;
```

As long as ATGOLD is FALSE, the WHILE statement will repeatedly execute the procedure GOCLOSER. Thus the procedure GOCLOSER will repeatedly print instructions that move the robot closer and closer to the gold until it has arrived at its destination.

We have now decomposed the problem of moving the robot to two subproblems: how to print instructions that move the robot closer (GOCLOSER) and how to test whether the robot is at the pot of gold (ATGOLD). Remaining from the first decomposition, we must also implement MAKEMAP and PICKGOLD. The two steps of decomposition have used sequencing and iteration.

As another example of decomposition by sequencing, consider GOCLOSER. We cannot move the robot forward unless we know that it is facing toward the gold. To handle this, we decompose GOCLOSER into two steps. First, print instructions that turn the robot in a direction that leads closer to the gold, and second, print instructions that actually make a move in the robot's facing direction. Thus we decompose to procedures FACEGOLD and MAKEMOVE, which perform these two steps and use sequencing:

```
PROCEDURE GOCLOSER;
BEGIN
      FACEGOLD;
      MAKEMOVE;
END;
```

We have now broken down the original problem in three decomposition steps. We have decomposed GETGOLD, GOTOGOLD, and GOCLOSER, and we are yet left with MAKEMAP, ATGOLD, FACEGOLD, MAKEMOVE, and PICKGOLD. We can diagram the decomposition as shown in Figure 1.3.

The list of unsolved problems is getting larger! Our hope is that the problems are also getting simpler. Certainly, we are acquiring a better understanding of the original problem.

As another example of decomposition by iteration, consider FACEGOLD, which turns the robot toward the gold. The robot can move from dot to dot