Data Structures and C Programs

73.8722
V217

Data Structures and C Programs

Christopher J. Van Wyk
AT&T Bell Laboratories
Murray Hill, New Jersey

9350009

ADDISON-WESLEY PUBLISHING COMPANY
Reading, Massachusetts • Menlo Park, California • New York
Don Mills, Ontario • Wokingham, England • Amsterdam • Bonn
Sydney • Singapore • Tokyo • Madrid • San Juan

To Claudia

This book is in the **Addison-Wesley Series in Computer Science**

Michael A. Harrison
Consulting Editor

Library of Congress Cataloging-in-Publication Data

```
Van Wyk, Christopher J.
   Data structures and C programs.

   Bibliography: p.
   Includes index.
   1. C (Computer program language)  2. Data
structures (Computer science)  I. Title.
QA76.73.C15V36  1988      005.7'3          88-3437
ISBN 0-201-16116-8
```

This book was typeset in Palatino, Helvetica, and Courier by the author, on an Autologic APS-5 phototypesetter and a DEC VAX® 8550 running the 9th Edition of the UNIX® operating system.

Reprinted with corrections April, 1989

DEFGHIJ-DO-89

Preface

One of the best things about computer science is that it offers the chance to build programs that do something, and at the same time to use interesting mathematics with which to study properties of those programs. This book is about some of the important tools that computer scientists use to study problems and to propose and choose among solutions.

Outline of the Book

Part I presents several fundamental ideas. These include abstractions like algorithm, data type, and complexity, and also programming tools like pointers, dynamic memory, and linked data structures. Chapter 6 presents a simple model of computer memory; the concrete details in this chapter suggest the source of some of the abstractions in Chapters 1 through 5.

Part II presents techniques to solve several general and important problems, especially searching and sorting. Chapter 12 shows how one might apply the material in Chapters 7 through 11 to solve a real-world problem; the emphasis is on building a program that can readily be changed to use different data structures and algorithms.

Part III surveys some advanced material about graphs and graph algorithms. The two chapters cover a lot of topics at a faster pace than Chapters 1 through 12, yet they offer only a hint of what lies beyond the scope of this book.

Each chapter concludes with a section called "Summary and Perspective," which highlights the chapter's important ideas and offers some thoughts on how they fit into the larger scheme of things.

How to Read this Book

It would be good to read this book with pencil and paper in hand. Pause at each problem as it is presented in the text; sketch your own solution before you see the one in the book. This will help you to appreciate the obstacles that any solution to the problem must face.

You can add even more to your reading by using a nearby computer to try your own solutions, and to test, modify, and experiment with the programs that are included in the text. Waiting for a slow program to finish can give you a visceral appreciation of what it means for running time to grow linearly or quadratically with input size.

Finally, your reading will be incomplete unless you do some of the exercises at the ends of each chapter. Many of the exercises reinforce the ideas in the text. Others ask you to extend results in the chapter to solve a new problem. Appendix D contains solutions to about one-fifth of the exercises.

C Programs

The programs in this book are written in C. Programmers who are new to C can consult Appendixes A and B, which contain a brief introduction to the C language and common C library functions; the programs in Chapters 1 through 3 should help you to pick up the essentials of the language.

Since C is a high-level language, it supports most of the abstractions that are important to writing good programs. At the same time, C reflects the architecture of contemporary computers and lets programmers take advantage of it, so a C programmer has a reassuring familiarity with the way a data structure is stored in computer memory. When I taught from an earlier version of this material at Stevens Institute of Technology, students who wrote in C generally understood the material better than those who wrote in Pascal, even though I used Pascal in lectures and sample solutions.

Anything in the text that is labelled "Program" was included directly from the source file of a computer program that was compiled and tested (under the 9th Edition of the UNIX operating system) before it could appear in the book. These programs are meant to illustrate computational methods, not to serve as models for software engineering. In general, they use global variables for simplicity, and contain few comments since they lie near text that explains them.

Some Perspective on Theory

Mathematical techniques figure prominently in the analysis of data structures and algorithms. Careful proofs of correctness give us confidence that our solutions do what they should, while asymptotic methods let us compare the running time and memory utilization of different solutions to the same problem. But mathematical methods are *means* with which to study, not ends in their own right. The programs in this book are meant to counteract the view that the mathematical analysis of data structures and algorithms is paramount.

Of course, some people believe that students of data structures and algorithms do not need to see programs at all. They contend that once you understand clearly the idea for a data structure or algorithm, you can easily write a computer program that uses it. They prefer to write problem solutions in high-level pseudo-code that omits many details. A few even go so far as to claim that "programming has no intellectual content."

I take strong exception to this dismissal of the importance of programming, which is, after all, the source of many interesting problems. I was surprised at how much I learned when I wrote the programs in this book. Sometimes, the final program bore little resemblance to the pseudo-code with which I had started; the program handled all of the details glossed over by the pseudo-code, however, and many times it was also more elegant. Seeing data structures and algorithms implemented also gives a better idea of how simple or complicated they are.

Another advantage of writing programs is that we can run them. This can give us insight into the performance of a particular technique. It can show us errors in our logical and mathematical analysis, or confirm it and give us more feeling for the practical importance of that analysis. Finally, by analyzing statistics gathered from programs, many researchers have been led to discover new data structures and algorithms.

Acknowledgements

I am grateful to many people who have influenced this book in some way. I learned a lot about putting theory into practice when I worked on projects with Brian Kernighan and Tom Szymanski. Al Aho and Jeff Ullman offered encouragement and advice when I conceived this book and started writing. Doug McIlroy and Rob Pike

gave me thoughtful comments on early drafts of several chapters. Jon Bentley and Brian Kernighan read the whole manuscript carefully; in fact, Brian waded through several versions.

I also offer thanks to the following people, whom Addison-Wesley recruited to review parts of the manuscript: Andrew Appel (Princeton University), Paul Hilfinger (University of California, Berkeley), Glen Keeney (Michigan State University), John Rasure (University of New Mexico), Richard Reid (Michigan State University), Henry Ruston (Polytechnic University of New York), and Charles M. Williams (Georgia State University); and to these people, who taught classes from the manuscript: Michael Clancy (University of California, Berkeley), Don Hush (University of New Mexico), and Harley Myler and Greg Heileman (University of Central Florida).

Murray Hill, New Jersey *C.J.V.W.*

Contents

Part I

Fundamental Ideas

1 Charting Our Course

The questions we ask when we study data structures and algorithms have their roots in practice: someone needs a program that does a job, and does it efficiently. The techniques we shall see in the chapters to come were discovered in the quest to create or improve a solution to some practical problem. To give some idea of the circumstances that often surround such discoveries, we shall solve a simple, practical, problem in this chapter.

The two solutions we shall see use only rudimentary programming techniques. Both solutions work, which is an important and good property. But both solutions also have serious limitations: the first is inconvenient for users, while the second takes longer and longer to run as its input grows. These limitations can be overcome only by using more sophisticated data structures and algorithms. Our reflections on these solutions offer some glimpses of important issues in the study of data structures and algorithms.

1.1 PROBLEM: SUMMARIZING DATA

The problem is to write a program with which to keep track of money in a checking account. We want to know both how money is spent on different expense categories (food, rent, books, etc.), and how money comes into the checking account from different sources (salary, gifts, interest, etc.). Following standard bookkeeping practice, we call both expense categories and sources of income *accounts*.

At this point we shall leave the exact details of the input unspecified; instead, we say merely that the data is presented as a sequence of lines, with each line specifying a *transaction*, an account together with an amount to be added to or subtracted from the balance in that account. Each line has the form

account amount

Different solutions can use different ways to designate accounts, tailoring the choice to the programmer's or the user's convenience. We do specify, however, that the output should have the same form as the input, with one line for each distinct account designation, and the amount on that line equal to the sum of the amounts on all input lines containing that designation.

For example, given as input the following six transactions:

```
salary 275.31
rent -250
salary 125.43
food -23.59
books -60.42
food -18.07
```

the program should produce the following output summary:

```
salary 400.74
rent -250
food -41.66
books -60.42
```

As a matter of fact, neither of the programs presented in this chapter accepts exactly this input, although Solution II comes close.

Problems like this arise in many situations. A solution to this problem could be used to maintain the balances in customers' charge accounts at a store; the output reports the amount owed in each account. It could also be used to follow inventories, with each account corresponding to a particular product, perhaps a dish served in a restaurant or a tool stocked by a hardware store.

The requirement that output be acceptable as input is not meant to dash creative efforts at report design, although it does leave us with fewer decisions to make. A program is often more useful if it can process what it produces. For example, given a program that solves this problem, we might use it to summarize checking account activity by the month; to arrive at an overall summary for the year, we would simply run the monthly summaries through the same program that produced them. The same idea applies to inventory control for a large corporation: if each restaurant or store in a region sends its summarized sales data to regional headquarters, and the summaries are in the appropriate format, then regional headquarters can summarize the data from all franchisees in the region and send the results to national headquarters.

1.2 SOLUTION I

In our first solution we adopt an input format expressly chosen to make our programming job simple: each account is designated by a non-negative integer less than *n*, where *n* is to be specified in advance. (Although this choice makes it simple to write the program, it is inconvenient for users, as mentioned in the chapter introduction.) For example, if *n* were five or larger, the following input would be acceptable:

```
0 275.31
1 -250
0 125.43
3 -23.59
4 -60.42
3 -18.07
```

This method for designating accounts suggests a natural data structure to use in a C program, since the first position in a C array is at index zero. We declare *balance*[ ] to be an array of length *n*, then store the balance of account *i* in *balance*[*i*].

We present our solution in a top-down fashion, beginning with a high-level view and refining the details to simpler steps until we reach a working program. We begin with the following outline:

```
(1) read and process each transaction line
(2) print a summary table
```

We elaborate Step (1) more fully as follows:

```
on each line,
    (1a) read two numbers—the account number and the transaction amount
    (1b) update the appropriate element of balance[ ]
```

Step (2) is simpler:

```
#include "ourhdr.h"

#define N 5
float balance[N];

void getlines() /* read and process each line */
{
    int account;
    float amount;
    while (scanf("%d %f", &account, &amount) != EOF)
        balance[account] += amount;
}

void printsummary() /* produce a summary table */
{
    int i;
    for (i = 0; i < N; i++)
        printf("%d %g\n", i, balance[i]);
}

main()
{
    getlines();
    printsummary();
    exit(0);
}
```

PROGRAM 1.1

Solution I to the problem in Section 1.1. See Appendix A for a brief introduction to the C language, Appendix B for a description of the library functions `exit()`, `printf()`, and `scanf()`, and Appendix C for the contents of the header file `ourhdr.h`.

```
as i goes from 0 through n−1,
    print i and balance[i]
```

The outline is now detailed enough that we can write Program 1.1. Program 1.1 uses `N` to denote the size of array `balance[]`, so we can change that size by simply redefining `N` if necessary. To keep the program simple, array `balance[]` contains `float`s; if we were dealing with real money (especially money that belonged to other people) we would want to be more careful about the precision with which amounts of money are stored.