

title: Problem Solving With Computers
author: Scragg, Greg W.
publisher: Jones & Bartlett Publishers, Inc.
isbn10 | asin: 0867204958
print isbn13: 9780867204957
ebook isbn13: 9780585348926
language: English
subject Electronic data processing, Problem solving—Data processing.
publication date: 1997
lcc: QA76.S393 1997eb
ddc: 005
subject: Electronic data processing, Problem solving—Data processing.

Problem Solving with Computers

Greg W. Scragg
Computer Science Department
SUNY Geneseo



Jones and Bartlett Publishers
Sudbury, Massachusetts
Boston London Singapore

Editorial, Sales, and Customer Service Offices
Jones and Bartlett Publishers
40 Tall Pine Drive
Sudbury, MA 01776
(508) 443-5000
(800) 832-0034
info @ jpub.com
<http://www.jpub.com>

Jones and Bartlett Publishers International
Barb House, Barb Mews
London W6 7PA
UK

Copyright © 1997 by Jones and Bartlett Publishers, Inc.
All rights reserved. No part of the material protected by this
copyright notice may be
reproduced or utilized in any form, electronic or mechanical,
including photocopying,
recording, or by any information storage and retrieval system,
without written permission
from the copyright owner.

Library of Congress Cataloging-in-Publication Data

Scragg, Greg W.
Problem solving with computers / Greg W. Scragg.
p. cm.
Includes index.

ISBN 0-86720-495-8
1. Electronic data processing. 2. Problem solving
processing. I. Title.

QA76.S393 1996

005dc20

96-21035

CIP

Acquisitions Editor: Dave Geggis
Production Editor: Martha Stearns
Manufacturing Manager: Dana L. Cerrito
Design: George McLean
Editorial Production Service: Kathy Smith
Typesetting: ICPC
Cover Design: Marshall Henrichs
Printing and Binding: Courier Companies, Inc.
Cover Printing: Henry N. Sawyer Company, Inc.

Printed in the United States of America
00 99 98 97 96 10 9 8 7 6 5 4 3 2 1

Contents

Preface	xxiii
Overview	xxiii
What This Course is	xxiii
. . . And what This Course is Not	xxiii
The Goal of <i>Problem Solving with Computers</i>	xvi
Myth, Mystique, and Computers	xxvi
“Using” Versus “Creating” Tools	xxvii
The Baggage of History	xxviii
The Baggage of Relevance	xxviii
The Learning Approach	xxviii
<i>Some Guidelines for Using Problem Solving with Computers</i>	xxx
Using the Related Material	xxx
Reading the Text	xxxi
Conventions and Layout of the Text	xxxi
Chapter Length	xxxi
The Use of Fonts	xxxi
Gender-Specific Terminology	xxxii
Acknowledgments	xxxiii
Chapter 0: Introduction	1
0.1 How do you Solve Problems?	1

What is a Problem?	2
What is a Solution?	3
Recognizing and Evaluating Solutions	3
0.2 Computer Science and Problem Solving	4
Some Problem-Solving Methods from Computer Science	5
0.3 Summary	8

Chapter 1: Using Tools: Word Processors (A Case Study)	9
1.0 Overview	9
Is Writing a Problem?	10
1.1 Describing Tools	10
Flexibility	11
Use or Function	12
Power Level	13
Objects, Tools, and Users	14
Granularity	14
1.2 A Word Processor as a Tool	15
Selecting Objects	17
1.3 Using Descriptions to Understand Actions	19
Granularity	19
Intermediate-Level Objects	20
1.4 Summary	22
Chapter 2: Tools and Computer Science	23
2.0 Overview	23
2.1 The Three most Important Computer Tools	23
Help	24
Undo	27
Save	28
2.2 And the most Important Self-Help Tool: Try it!	31
The Scientific Method	31

2.3 Metadiscussion: Thinking About the Previous Discussion	32
Analogies	33
Abstraction	33
2.4 Participants in the Use of Tools	34
Ethics is a Problem?	35
The First Rule of Computer Ethics	35
2.5 Summary	37

Chapter 3: Documents as Objects	39
3.0 Overview	39
3.1 The System and High-Granularity Operations	39
Operations on Documents	39
System Commands	41
Finding the Commands	41
3.2 Understanding Memory	42
Classifying Long-Term Memory	44
3.3 Understanding Files	45
Types of Objects	46
Organization	47
Naming Files and Other Objects	48
The Desktop as an Object	49
Applications as Data	49
3.4 Tools and their Creators	50
Ownership: Who Actually Owns Software?	50
Intellectual Property	51
Common Misconceptions	52
3.5 Summary	53
Chapter 4: Sharing and Communicating Information	55
4.0 Overview	55
4.1 Shared Data	56
4.2 File Servers	56

Computer Networks	56
Danger: Viruses	58
4.3 Mail Systems	60
The Post-Office Model for Mail Tools	60
Mail Tools Based on the File System Model	61
Email as a Word Processor	62
Hybrid Tools	62

Thinking About Mail Systems	63
Suggestions for Using Email Systems	64
4.4 Selecting the Best Tool	64
4.5 Mail, Problem Solving, and Computing	65
4.6 Questions as Problem-Solving Tools	66
The Question as Teamwork	67
Extending the Concept of Communication	67
4.7 Privacy Issues of Electronic Mail	67
Junk Mail	68
Electronic Harassment	68
Privacy of Communication	69
4.8 Summary	70
Chapter 5: Representing Information	71
5.0 Overview	71
5.1 Three Views of the Structure of Textual Information	71
More Complicated Structures	72
Dealing with Multiple Views	73
5.2 Information Structure as a Problem-Solving Tool	74
Whose Problem is it, Anyway?	74
5.3 Visual Tools for Emphasizing Structure	76
<i>Ad hoc</i> Techniques for Organization	76
Fonts	76
Styles	77

Size	78
Additional Structuring Tools	78
5.4 Selecting Tools	79
5.5 Structure and Computerized Documents	80
Wrap Around	80
Selection	81
Invisible Characters	81
Null or Empty Objects	82
5.6 Describing Text and Characters	83

Serifs	84
Spacing	84
Ascenders and Descenders	86
Special Fonts and Special Characters	86
5.7 Syntax and Semantics of Natural Language	87
5.8 Plagiarism Made Easy	88
5.9 References	89
5.10 Summary	89
Actions on Parts of a File	89
Chapter 6: Hierarchical Organization of Information	91
6.0 Overview	91
6.1 Problems in Data Organization	91
Reading a Document	92
Updating or Maintaining a Document	92
Creating a Document	92
6.2 Hierarchical Organizations	93
Garden-Variety Trees	93
Family Tree	94
Sports Playoff	96
Corporate Organization	97
Computer Folders	98
Decision Trees	99
6.3 Outlines	100

The Outlines as a Tool for the Reader	101
The Outline as a Tool for Creating a Document	102
Outlines and Computerized Documents	104
6.4 Trees and Computer Science	106
The Definition of <i>Tree</i>	106
Trees and Searching	107
6.5 Structures Other Than Hierarchical	108
Lists	108

Tables	109
Graphs, Pointers, and Multiple Organizations	109
6.6 Summary	109
Chapter 7: Models, Visualization, and Pictures	111
7.0 Overview	111
7.1 The Advantage of Pictures	112
7.2 Objects and Images	114
7.3 The Models Define the Tools	115
Paint Tools	116
Draw Tools	118
Comparing the Models	119
Selecting the Correct Tool for Visualization	121
7.4 Extending Text Concepts to Visualization Tools	122
7.5 Documents with Mixed Forms	123
Mixed-Form Documents	124
7.6 When is a Picture Useful?	124
Times Not to Use Pictures	125
7.7 Related Visualization Tools	125
Clip Art	125
Image Viewers	126
Scanners	126
Facsimile Machines	126
7.8 Computers and Pornography?	127

7.9 Summary	127
Chapter 8: Relations, Tables, and Databases	129
8.0 Overview	129
8.1 Uniform Data Structures	129
Lists and Sets	130
Relations	132

Tables	133
Using Relations	135
8.2 The Database: A Tool for Representing and Manipulating Relations	136
Operations on Relations and Tables	138
Analogies with Other Tools	139
8.3 Living Documents	140
The Document Life Cycle: Designers and Users	141
8.4 Using a Database Tool	143
Granularity	143
Techniques for Describing Relations	143
More Complex Situations	144
8.5 Ethics of Data Collection	144
Accuracy of Databases	145
8.6 Summary	146
Chapter 9: Functions and Calculated Values	147
9.0 Overview	147
9.1 Introduction to Functions	147
9.2 Computability of Functions	149
9.3 Reasons for Using Functions	153
Save Time	154
Reduce Errors	154
Deal with Change	154

Terminology for Functions	155
9.4 A Tool for Representing Functions: The Spreadsheet	157
Historical Anachronisms and the Representation of Functions	159
Expanding the Basic Set of Operations	161
Examples of Uses of Spreadsheets	162
9.5 Thinking About the Tool	164
Analogies to Other Tools	164
Granularity	164
Spreadsheets Versus Databases	165
9.6 Summary	165

Chapter 10: Functions and Relations as Problem-Solving 167
Tools

10.0 Overview	167
10.1 Rules of Thumb as Relations	167
Approximate Answers as a Problem-Solving Method	168
Spreadsheets as Tools for Developing Rules of Thumb	170
Rules of Thumb for Verification	173
10.2 Calculations with Multiple Answers	175
Complex Relations	175
Spreadsheets and Tables	176
10.3 Thinking About Functions	178
Dealing with Representation	178
Dealing with Generality	178
10.4 Techniques for Describing Functions Using Spreadsheets	179
10.5 Summary	181
Chapter 11: Algorithms	183
11.0 Overview	183
11.1 The Definition of Algorithm	183
Exploring the Definition of Algorithm	186
Algorithms, Computer Instructions, and Computer Science	189
11.2 Arithmetic Expressions as Algorithms	189

Unambiguous Representations can Force Unambiguous Interpretations	190
Arithmetic Expressions and Trees	196
Nonarithmetic Expressions	196
11.3 Summary	199
Chapter 12: Generalizing Problem Solutions	201
12.0 Overview	201
12.1 The Role of Algorithms in Problem Solving	201

Precision	202
Reuse	202
Debugging	203
Sharing the Labor	203
12.2 Algorithms and Empirical Results	204
Experimental Determination of Order of Evaluation	204
Rules of Thumb and Algorithms	206
12.3 When Algorithms Fail	207
Infinite Algorithms	207
Poorly Ordered Algorithms	209
12.4 Exploring Possible Scenarios	210
12.5 Summary	214
Chapter 13: Programming in Applications	217
13.0 Overview	217
13.1 The Programming Process	217
13.2 Problem Definition	222
Ask Questions	223
Input-Output as Defining Factors for Problem Definition	223
Other Definition Tools	224
13.3 Algorithm Development in Context	224
Use Top-Down Design	224
Use Pseudocode	225

13.4 Coding	226
13.5 Documentation	228
Document as you Code	228
Document for the Concerned Parties	228
Some Documentation Guidelines	230
13.6 Debugging, Testing, and Verification	230
13.7 Physical Appearance	231
Whitespace	231
Comments	232
Organization and Readability	232
13.8 Summary	234

Chapter 14: Dealing with Errors	235
14.0 Overview	235
14.1 Sources of Error	235
14.2 Removing Error	237
The Debugging Process	237
Debugging and the Scientific Method	237
14.3 Anticipating Errors or Anti-Bugging	238
14.4 Testing and Verification	240
Assume that your Program is in Error	240
Plans	241
Test Data	242
Check Points	244
Seeking Help	245
14.5 Classes of Error	245
Syntax or Compilation Errors	246
Run-Time Errors	247
Machine Error	248
Logical Errors	249
14.6 Misconceptions About Good Programming	249
So your Program is More Efficient, is it?	250
14.7 Summary	251
Chapter 15: Iteration: Replicated Structures	253
15.0 Overview	253

15.1 Iterative Data Structures	253
Table Organizations for Data	254
15.2 Control Structures	255
Iteration and Spreadsheets	257
Iteration is Ubiquitous	258
15.3 Iteration as an Algorithmic Tool	260
Toy Problems and Algorithm Development	261
Iteration and "What-if" Problems	261

Iteration and Nonnumeric Series	263
Double Iteration	264
15.4 Programming Techniques and Iteration	265
Implications of Iteration on the Design of Algorithms	265
Scope and Abstraction	267
Iteration and Debugging	267
Execution	268
Documentation	268
15.5 Summary	269
Chapter 16: Iteration: Replicated Algorithms	271
16.0 Overview	271
16.1 Sequences and Recursively Defined Lists	271
Simple Sequences	271
Nonnumeric Series	273
16.2 Recursively Defined Functions	274
A Hybrid Example	280
16.3 Relative Versus Absolute Addressing	282
16.4 Recursion and “What-if” Problems	284
16.5 Iteration as a Model of Analytic Integration	286
16.6 Summary	287
Chapter 17: Conditional Actions	289
17.0 Overview	289

17.1 Algorithms with Alternatives	289
Anatomy of a Conditional	290
Representation of Questions	291
Conditionals in a Spreadsheet	292
17.2 Arithmetic Relations as Functions	295

17.3 Exploring Conditionals	297
Uses for Conditionals	298
Invisible Conditionals	299
Conditionals with Nonarithmetic Operands	301
17.4 Conditional in More Complicated Algorithms	302
17.5 Summary	305
Chapter 18: Applied Logic	307
18.0 Overview	307
18.1 Boolean Logic	307
Conjunction (And)	308
Disjunction (Or)	309
Negation or Not	312
Boolean Operations as Functions	313
Boolean Operations and Nonnumeric Arguments	313
18.2 Boolean Algebra	314
Complex Truth Tables	315
Manipulating Boolean Expressions	316
18.3 Questions with More Than Two Possible Answers	317
18.4 Logic in Applications	320
Tables and Conditionals	320
Databases and Logical Expressions	321
Manipulation of Relations	322
Testing, Debugging, and Representing Complex	323

Logical Structures

18.5 Conditionals and Hierarchical Structures	324
18.6 Summary	326
Chapter 19: Abstraction, Abbreviation, and Macros	327
19.0 Overview	327

19.1 Abstraction	328
Abbreviations	328
Abbreviations in Computing Systems	329
19.2 Abstraction and Procedures	333
Built-in Procedures	333
Grouping Actions	335
Predefined Combinations	336
Other Examples	336
19.3 Building New Macros	337
Variations	338
Templates	339
Scope	339
19.4 Macros and Abstractions	340
Named Actions	340
Documentation	340
19.5 Summary	341
Chapter 20: Visualization Revisited	343
20.0 Overview	343
20.1 Visualization as Data Reduction	344
20.2 Classes of Problems	345
Simple Comparisons	345
Problems of the Whole	348
Trends	350

A Quick Review of the Cartesian Coordinate System	352
Combinations of Values	353
Functions	354
Irregularly Distributed Data	355
20.3 Misleading Charts	359
Misleading Proportion	359
Rules of Thumb for Using Visualization Tools	360
20.4 Charts as Tools for Understanding Mathematical Relations	361

Exponential and Logarithmic Values	361
Graphing and the Calculus	362
20.5 Summary	364
Chapter 21: Graphs and Hypermedia	365
21.0 Overview	365
21.1 Graphs	365
Embedded Organizations	365
Limitations of Graphs	367
21.2 Related Tools	367
Multiply Ordered Databases	367
Indices and Tables of Pointers	368
Voice Mail	370
21.3 Dynamically Linked Documents	370
Dynamic Links	371
Interaction	371
Visualizing Hypermedia	371
Familiar Examples of Hyper-Documents	371
Common Pointers	372
21.4 Navigation in Hypermedia	373
Linear Structure	373
Hierarchical Structure	374
Generalized Pointers	375
21.5 Hypermedia Applications	375

The World Wide Web	375
Gopher	376
Hypermedia Programming Environments	377
Multimedia	377
21.6 Search Strategies	377
Standardized Starting Points	378
Superimposed Structure	378
Hill Climbing	379
Bookmarks and Initial Leaps	379
Understanding Forward and Backward	379

21.7 Search Engines	380
21.8 Summary	382
Chapter 22: Telecomputing	383
22.0 Overview	383
22.1 Introduction to Telecommunication	383
Levels of Network	384
The “Information Superhighway”	384
22.2 Common Forms of Telecomputing	385
The World Wide Web	385
Electronic Mail	386
Bulletin Boards and News Groups	388
Remote Access	389
Document Retrieval	389
Database Searches	390
Specialized Applications	391
Network Applications	391
22.3 Advantages of Network Access	392
A Familiar Example	393
22.4 Variations on Network Access	393
Commercial and Academic Varieties of Networking	393
Means of Access	394
22.5 Ethical Considerations of Telecomputing	395
Freeware Versus Shareware	395

Breaking and Entering	397
22.6 Summary	398
Chapter 23: Hypermedia Construction	401
23.0 Overview	401
23.1 Dynamic Links	401
Construction of a Link	402
One Task, Two Tools	402

Compilation Applications	403
Interpretation Applications	404
23.2 Visualizing Data Transitions	406
Modeling the Document	406
23.3 Logical Types of Links	407
Linear Structures	407
Hierarchical Structures	408
<i>Ad hoc</i> Pointers	408
23.4 Guidelines for Creating Links	409
23.5 Visual Link Conventions	411
Buttons and Structure	411
Placement of Buttons	413
Variations on Simple Links	413
23.6 Problems of Hypermedia	414
23.7 Another “Hyper”: Hypercubes	415
23.8 Summary	416
Chapter 24: Algorithms Revisited	419
24.0 Overview	419
24.1 Formal Languages and Written Algorithms	420
Algorithms for Navigation	420
24.2 New Algorithmic Representations	422
Objects and Messages	422
Examining an Existing Algorithm	423

Some Other Primitive Tools	424
Old Structures in a New Environment	425
24.3 Assignment Statements	425
Expressions	426
Visible and Invisible Storage Locations	428
Scope	429
24.4 Algorithms and Visualization	429
Algorithms that Draw Pictures	430
Coordinate Systems	430

24.5 Programming Revisited	431
Other Capabilities	432
Old Techniques in a New Environment	432
Testing and Debugging	433
Build Incrementally	433
Documentation	433
24.6 References	434
24.7 Summary	434
Chapter 25: Control Structures Revisited	437
25.0 Overview	437
25.1 Iteration	437
25.2 Conditionals	440
25.3 Abstraction	442
Documentation	442
Scope and Delimiters	443
25.4 Procedural Abstraction	444
Basic Procedures	444
Communications to Procedures	445
Communication from Procedures	447
25.5 Recursion	449
Recursion and Pictures	450
25.6 Programming Revisited	452
Structure Suggestions	452

The “Drawability” Test	452
Documentation Guidelines	452
25.7 Summary	453
Chapter 26: Where to from Here?	455
26.0 Overview	455
26.1 What will Computing Look Like in the Future?	455

The Past as a Window on the Future	456
Users Must Adapt	457
26.2 Learning to Use New Software Tools	458
On Reading a Manual	459
26.3 Other Types of Common User-Oriented Software	460
Personal Finance Programs	460
Tax Calculations	461
Networking Software	461
Office Management	461
Databases and Information Archives	461
High-End Graphics	461
Sound	462
Specialized Personal Software	462
Specialized Career-Oriented Software	462
The Bottom Line	462
26.4 What Other Courses Should I Take?	462
Computer Science and Computing	463
Other Fields and Computing	464
26.5 Will Computers be Used for Good or Evil?	464
26.6 Summary	466
Index	467

Preface

Learning without thought is labor lost.
CONFUCIUS (*Analects*)

Pointers1

Lab manual: Preface to the Lab Manual

Overview

Problem Solving with Computers represents a nonstandard introduction to computer science. The focus throughout is on the methods of problem solving, rather than on the hardware or software tools employed as aids for that problem solving.

What This Course is

Problem Solving with Computers (PSC) is a first course in computer science. It is about problem solving in particular, solving problems using computers. *PSC* aims to improve your problem-solving skills by introducing the methodology of computer science. With better problem-solving skills, you will be able to use a variety of computer tools *advantageously* in the solution of problems, including many problems that you will certainly face in the future.

... And what This Course is Not

The four most common forms of the “first course in computer science,” are typically called:

1. *Computer applications*,
2. *Computer literacy*,

1. *Problem Solving with Computers* has an accompanying lab manual and tutorial software, as described in Section 4.1 of the preface. To help

you use the tools together, each section is cross-referenced with pointers to the others. If you have these tools, you are urged to follow the pointers.

3. *Survey of Computer Science*, and
4. *Introduction to Computer Science* (using some particular language).

Problem Solving with Computers differs from each of these formats. A brief description of these other approaches and a discussion of the differences will help clarify the goals of the approach used here.

Computer Applications.

Many first courses, which may have titles such as *Microcomputer Applications* or *Introduction to Computers*, introduce students to computation by introducing selected common applications such as word processors or spreadsheets. To some, this approach may appear to be the one most closely related to *PSC*. Certainly, many students take *Problem Solving with Computers* with the intent of learning the material typically covered by *Computer Applications* courses. And they will certainly gain most (or all) of those skills. However, the emphasis, approach, and content of *PSC* all differ from those of *Computer Applications* approaches.

It is not the goal of *Problem Solving with Computers* to teach any specific computer applications. That is, this is not a course about *Microsoft Word®*, or *WordPerfect®*, or *Lotus 1-2-3®*, or . . . (fill in any specific word processor, spreadsheet, or other product name). Those are specific tools that support the problem-solving process; they are not approaches to problem solving. Just as a driver's education class is not a course about Fords, and a cooking class is not a course about Farberware®, this is not a course in any specific application tool. Yes, you will use specific tools, but they are the vehicle not the object of study. The material focuses on the commonality of the tools, not the distinctions of the products. By the time you have finished the course, you will have experience with a variety of computer tools. More importantly, you will have the ability to learn to use many more tools on your own. Thus,

although *PSC* fills the goals of the *Computer Applications* approach, it also prepares you for the new applications you will need later. In *PSC*:

The emphasis is on the problem-solving process itselfnot on the specific tools.

The emphasis is on generalizable conceptsnot on specific implementations.

The quantity of instruction is lowthe quantity of practice is high.

The general approach is based on the important principles of computer science.

Computer Literacy.

The term *computer literacy* means many things to different people. For some, it refers to one of the other introductions described here. Some versions of this course may be described as *History of Computing* or *Vocabulary of Computers*. These courses help students understand the impact computers have had on contemporary society. Some introductions seem to concentrate on learning interesting facts about computers, rather than learning to use them:

Herman Hollerith invented the punched card to help with the 1890 U.S. Census, and went on to found IBM . . .

Although this is an interesting historical fact, it does not help you use a computer. It is not necessary to know the relative roles played by Henry Ford and Ransom Olds in order to drive a car. Sometimes, this type of information does help provide a foundation for understanding. *Problem Solving with Computers* introduces such historical facts only when they help explain modern usage.

Historical facts may be introduced to explain "why things are the way they are" or because they provide an amusing diversion, but never as important facts to be memorized. Even then, facts are not introduced until they are needed. Similarly, undirected vocabulary and nomenclature (e.g., learning the definitions of, and distinctions between, *arithmetic logic unit*, *central processing unit*, and *control unit*) are unenlightening to a casual user. Yes, *PSC* does introduce vocabulary where it helps explain specific problem-solving techniques. Obviously, names are importantthey allow you to refer to a previously discussed conceptbut unnecessary vocabulary is kept to a minimum. The intent here is that you should never wonder, "Now, why are they telling me that?" Instead, all new and strange material should generate the question, "How can I use this information?"

Introduction to Computer Science.

Most *Introduction to Computer Science* courses (generically called *CS1*) are aimed at prospective computer science majors and focus on the concept of *algorithm*. They almost universally involve a so-called *higher-level language* such as *Pascal*, *Modula*, *C++*, or *Scheme*. These languages are the primary software tools used in computer science for constructing algorithms. Often it appears to students as if these languages are the primary focus of the course. Such emphasis is not surprising: at one time (not very long ago), common wisdom predicted that every educated person would need to know a higher-level language. Today more powerful and easy-to-use applications have replaced higher-level languages as the primary computer tools for most non-computer scientists.² But the

change in tool usage in no way implies that the discipline of computer science has nothing to teach the non-major. Although *Problem Solving with Computers* does involve algorithms and even introduces some higher-level language concepts, its primary focus remains the problem-solving process, and its most frequent software tools are popular software packages.

Survey of Computer Science.

The newest form of “first course in computing” attempts to provide an overview of computer science in terms of the subdisciplines of the field. Such courses help the student understand what

2. Historical note: the description *higher level* can be misleading. The so-called higher-level languages are much lower-level tools than the newer tools that you will use in this course. Higher-level languages are only higher than the tools that came before them chronologically. At the time of their invention, higher-level languages represented a major improvement in human-oriented design, and hence the name.

computer science is and what computer scientists do. These courses introduce the major topic or subject areas of interest to computer scientists and provide a good overview of what computer scientists do. *Problem Solving with Computers* focuses not on the problem domains, but on the problem-solving techniques and methods used by computer scientistshow computer scientists do what they do. Many of these techniques are exactly the same ones covered by a typical *Introduction to Computer Science* course. But, computer science is a problem-solving discipline. Its techniques are applicable in many environments and to the users of many kinds of tools.

The Goal of Problem Solving with Computers

Problem Solving with Computers focuses on the problem-solving process, using techniques that are common to many applications. *PSC* students often become more adept at the use of specific software tools than students who have had a computer applications course focusing on those tools. Why? Because they learn how to use the tool to advantage. No matter what tools you learn initially, new problem-solving environments will dictate that you use other tools in the future. The physics department may want you to use one word processor and the English department another. Certainly, the tools used by any potential future employer are not dictated by those used at any specific college (nor vice versa).

In fact, any specific application you use during this course will be obsolete by the time you graduate. At the very least, there will be a new version of the specific tools. In all likelihood, some of the applications will not even exist, having been replaced by a better competitor, or even a totally new kind of tool. No course that focuses on specific current products can teach you to use future products. You will always need to learn new tools; that is a given. And you certainly will not want to take a new course for each one. *PSC* prepares you for learning to use those new tools on your own.

Myth, Mystique, and Computers

To make effective use of a tool, you must feel comfortable with it. Unfortunately for many people, the comfort level is reduced by the many myths that have evolved about computers and computing. Feeling comfortable does require that you have a basic and true understanding of the tool. *PSC* attacks such myths as:

Computers are only for scientists and mathematicians.

Computers are difficult to use.

Computers can only process numbers.

Computers can destroy your life's work.

With computers, any incorrect action by a human can be a disaster.

Computers are somehow more mysterious than other tools.

Computers are big and dumb.

Computers may take over the world.

Using computers requires a different set of ethics than other activities.

The focus on problem-solving helps sort out the myths and remove the mystique not by studying the internal details of the machine, but by using computers as problem-solving tools.

“Using” Versus “Creating” Tools

One of the major developments of the Industrial Revolution was the *machine tool*, a machine that can make a tool (or another machine). This development enabled not just a speedup but a geometric³ speedup in the development of new tools. With the development of machine tools, people no longer had to make every tool they needed; they had tools that could make tools. This capability blurs the distinction between tools and the objects they manipulate. The machine tool creates a tool. That tool is the object manipulated by the machine tool.

In the information age, *programming* is the electronic equivalent of the machine tool. Computer programs are information-processing tools that can help make information-processing tools. As general purpose problem-solving tools, computers can be made to address almost any problem. Programming creates specialized problem solvers from general problem solvers by using the information specific to the problem at hand. A word processor is a program written by a human. To that human the word processor is an object; to most users, it is just a tool.

Much of the mystique that has grown up around computers actually concerns the concept of programming. It is the activity of programming that many people find difficult. Fortunately, it is not necessary to be a tool builder to be a tool user. You do not need to be an automotive engineer to drive a car. Similarly, you do not need to be a computer scientist, or even a computer programmer, to use

the tools that others have built. This course concentrates (especially at the beginning) on problem solving, using the tools that others have built for you. Discussions of programming concepts are restricted to domains that are immediately useful to the reader. This text includes no material on programming⁴ in classic higher-level, assembly, or machine languages.

3. An *arithmetic series* is a numeric series such as 2, 4, 6, 8, A *geometric series* one of the form: 1, 2, 4, 8, 16, . . . progresses considerably faster than does an arithmetic series. A geometric speedup is a very significant speedup. Both forms of series appear repeatedly in computer science. Section 16.2 contains examples illustrating just how great this difference is.
4. This text (starting in *Chapter 11: Algorithms*) uses the word *programming* to refer to actions in end-user applications that have many of the traditional characteristics of programming. When you create programs for the exercises in the later sections of this course, you will use the same mental skills, but without many of the problems that give traditional programming languages a bad reputation.

The Baggage of History

A second source of myths about computing comes from looking backwards. Like any tool, computing has evolved to fit the needs of its users. The earliest major users of computers were mathematicians and scientistsusers who were not intimidated by complicated mathematical interfaces. Fortunately, computers have become easier to use with each passing year. Most personal computers now have *graphical user interfaces (gui)*. This simply means that the user controls (*interfaces with*) a machine by manipulating pictures (*graphics*) of objects. Gui computer actions are direct analogs of their real-world counterparts. The user selects items by pointing at them (with a mouse) and gets rid of objects by dragging them to the trash, and so on. It is no longer necessary to memorize arcane commands for simple tasks. Instructing a computer to perform most tasks is a matter of selecting the correct actionnot of remembering the instruction. Modern screens are *wysiwyg* (pronounced wizz-ee-wig), or “*What you see is what you get!*” Users no longer view a mathematical description of their data, but the data itself. The status of your work is almost always directly visible. What some people fear is not the computers of today, but those of yesterday. *Problem Solving with Computers* focuses on the selection of appropriate commandsnot on the details of those commands.⁵

The Baggage of Relevance

Any subject that is foreign to you will seem difficult to comprehend. Concepts are easy to comprehend only when they can be understood in context. Vocabulary common in the computer world sounds like babble to outsiders. Computer vocabulary is better understood and far less mysterious in useful contexts. This text introduces new terms only when they become useful for solving a problem. There is no reason to fear the alphabet soup that seems to pervade many computer discussions.

The Learning Approach

If I were founding a university I would first found a smoking room; then when I had a little more money in hand I would found a dormitory; then after that, or more probably with it, a decent reading room and a library. After that, if I still had more money that I couldn't use, I would hire a professor and get some text books.

STEPHEN BUTLER LEACOCK

Mr. Leacock's unfortunate predilection for smoking aside, his general observations are important: people learn by interacting and working with ideas. He places more

5. Although the text attempts to be application independent, it does assume a modern gui interface such as found in *Macintosh* or *Sun* computers or the *Windows* system popular on *IBM* compatibles.

importance on interaction and discussion than on either books or professors. Reading and listening to lectures, while important tools, are subordinate to experience and practice. The most important aspects of learning are doing or practicing. You can learn more by doing a thing than by being told about it. In this course, this means that you will use tools early and often. It also means that you will use computer-aided tools for learning. The learning tools not only teach factual knowledge; they provide exercise in common aspects of other computer tools. There will also be “incidental” practice along the way. For example, you may use *electronic mail* to communicate with the instructor or to turn in your homework, or use electronic *file servers* to obtain supplementary material. Such incidental practice provides a nearly painless path to an expertise in the low-level details of computer interaction.

A quarter century after Mr. Leacock made his comments came a similar, but bolder, statement:

American universities are 500 years out of date.

MARGARET MEAD (*Lecture at the University of California at Riverside*)

Why, she asked, should a student listen to a lecture⁶ when reading a book would be better? Lectures proceed at the rate chosen by the lecturer; reading proceeds at the rate chosen by the reader. Text can be reread; lectures are seldom “respoken.” Sections of a text that are already familiar can be skipped (in lectures, familiarity induces sleep).

This text pushes this concept a step further. I claim that textbooks are also out of date. Learning comes best through practice far better than from reading or listening. Until recently, books were the only medium through which mass education could be delivered. But texts do not deliver practice or experience. Therefore, the text (and hopefully, lectures) is kept to a minimum. You will be asked to practice problem solving extensively, which incidentally requires that you also interact with computers extensively. *Problem Solving*

with Computers includes many computerized lessons, which can substitute for some of the traditional lecture notes and reading assignments. The lessons provide practice computing from the very beginning of the course. You should do all assignments, and more. Practice using the tools as you see them.

Many lessons ask you to “jump in and swim,” even before telling you much about swimming. Fear not. There is little risk of drowning. The experience is well worth the effort. The lessons will interact with your experiences to improve your understanding.

Please extend the concept of interaction one step further. Although every course has lectures, many instructors of this course do not use that term in the

6. The term *lecture*, after all, does come from the Latin *lectura*: to read.

traditional sense of the instructor standing before the class and filling students' heads with knowledge (probably a poor technique in general, but especially poor in activity-heavy areas, such as athletics, music, and computer science). I encourage you to ask questions in "lecture." Lectures can often be dynamic interactions in which the subject changes at the request of the students.

Some Guidelines for Using Problem Solving with Computers

Using the Related Material

In addition to this text, *Problem Solving with Computers* has two related packages of learning tools:

a laboratory manual, and

a series of *Gateway Labs*.

The text can be used alone, but together these three tools form an integrated three-part package. Certainly, many classes will use alternative material. Such material will cover similar concepts, but may be better tailored to the local needs at your institution. For example, the manufacturers of many computer tools provide their own gateway-like tools.

The Laboratory Manual.

The lab manual serves as the "tour guide" for the learning process. As you would expect from its name, it contains laboratory assignments and instructions. It also contains pointers to the *Gateway Labs*, basic introductions to specific applications, and homework assignments.

The Gateway Labs.

These are a collection of tutorial programs designed to get you started on various aspects of solving problems with computers. The tutorials provide information, but more important, they ask you to use that information. They are based on the premise that you will

learn more if you perform a given action than if you only read about it. Each tutorial asks you to practice the concepts and provides feedback about your success or failure in applying the concepts. The reward for the extra effort spent on the tutorials is a reduction in the time required for reading. These labs are available through the publisher's home page (at <http://www.jpub.com>).

Throughout the book, you will find pointersreferences to the lab manual and to the *Gateway Labs*. These pointers direct you to relevant material in the other parts of the package.

Science and Math Concepts.

This text is aimed at the general reader. However, some material does require a little more mathematical or scientific experience than does the text as a whole. Similarly, some segments have more value to science and math students. Such segments are marked with the symbol shown at the left.

Reading the Text

Tools for learning that you will invoke while studying problem solving include:

reading the text,

stopping to think,

running the *Gateway Labs*,

stopping to think,

completing exercises (both explicit stop-and-think, and turn-in exercises),

doing laboratory assignments,

stopping to think,

completing homework assignments, and

thinking about the material.

Yes, reading and doing exercises are tooltools for learning. Do not read the text as material that is to be “gotten through.” Do stop often. Stop and think. At some points, the text contains explicit instructions to do so. Stop and think at least that often.

Conventions and Layout of the Text

Chapter Length

The chapters of this text are relatively short to encourage you to

stop reading, both to “go do” and to think. Thus, this text has more, but shorter, chapters than do most similar texts. In fact, there are 27 chapters about one for each lecture in a typical course. The idea is that no block of reading should be too long. Experiment with ideas as soon as possible after you see them in the text. If possible, read with a computer nearby; interrupt your reading and try out what you have read as soon as you read it.

The Use of Fonts

The word *font* refers to the characteristics of the typeface. You may already have noticed that the font in this text seems to change periodically. These changes are not random. Changes in font provide hints about the usage of specific words. Although it is not necessary to understand the font usage to understand the text,

such understanding does provide an additional tool to help you disambiguate some references. Each of these uses represents a standard convention within computer science, mathematics, or other fields. Specific font conventions in this text include:

Normal Text.

Most text is written in this font (called “Times” as in *New York Times*), which is very common in microcomputer environments.

Definitions.

Words introduced for the first time or being defined in the current sentence are italicized. Thus, this section opened with “The word *font* refers . . .”

Title and References.

Titles of written works also use the standard convention of italicizing the title of major written works or quotations excerpted from another text. For example, *For Whom the Bell Tolls*.

Computer Words.

Descriptions of computer input and output often lead to awkward sounding sentences. The instruction: “Type `test` in the window `TestWindow`,” has a sort of *Jabberwocky* sound. Does it say there is a “type test” in the window? Actually, `test` is the word for you to type; `TestWindow` is the name of a window on the computer screen in which you should type `test`. Words used with specialized computer-based meanings are written in a simple-but-distinct font. The font change implies “This word is not used in its conventional English sense.” Without the font change, the sentence may be difficult to understand. In other contexts, authors use quotation marks around the “`test`”. Unfortunately, in the context of computer instructions, that solution is also ambiguous. It is not clear if the quote marks are something to be typed, or if they only

delimit what material to type. The font used here for computer words is called Helvetica (from *Helvetica*, the ancient Latin name for Switzerland).

Product Names.

The names of specific computer products are both computer words and titles. Product names create the same sort of hard-to-understand sentence. For example, *Word* is a word-processing program. You might say “I use Word,” but that sounds a bit odd unless the listener knows that *Word* is a product name. The font used for these, italicized Helvetica, reflects both aspects.

Gender-Specific Terminology

Wherever practical, I have tried to use gender-neutral terms. Where clarity required gender-specific pronouns, I have usually chosen the feminine “she” and “her” rather than the more traditional masculine “he,” “his,” and “him.” I did use the

masculine where required in quotations, examples, and references involving named or specific males. Some may think it surprising that this seems to have resulted in rough parity among gender-specific terms.

Acknowledgments

I would like to thank the many people whose efforts have contributed to this text: the students in the course “problem solving with computers at SUNY Geneseo” who put up with preliminary versions and provided valuable feedback; Indu Talwar of Geneseo’s computer science department who used preliminary versions in her own classes and provided feedback from her perspective as a faculty member; to the other members of the department who put up with me while I worked on a continuous stream of revisions; to G. Michael Schneider, Rhys Price-Jones, and Charles Kelemen who reviewed the text, providing many useful suggestions; and to the staff of Jones and Bartlett who shepherded the book through the various stages of the production process, particularly David Geggis, Carl Hesler, Martha Stearns, Rich Pirozzi, and Tanya Barrett, along with Kathy Smith, editorial production service.

References.

Many chapters of *Problem Solving with Computers* include annotated references. Do not think of these references as required reading, but as pointers to additional material that may be useful.

Definition of the Field.

The commonly accepted definition of the discipline of computer science was put forth in:

Denning, P., D. Corner, D. Gries, M. Mulder, A. Tucker, A. Turner, & P. Young. *Computing as a Discipline: Report of the ACM Task Force on the Core of Computer Science*. New York: ACM press, 1988. (Portions reprinted in *Communications of the ACM* 32(1)

(January 1989) and *Computer* (March 1989).

Quotations in this text were “cut and pasted” from the following sources:

*Correct Quotes*TM (a computer application) copyright 1991 by WordStar International, Inc.

The Columbia Dictionary of Quotations (computerized version). New York: Columbia University Press, 1993.

International Thesaurus of Quotations. Middlesex, England: Penguin Books, 1973.

Additional Information.

For more information and materials including the *Gateway Labs* please visit Jones and Bartlett's home page (at <http://www.jbpub.com>). You may also contact the author with your comments, suggestions, questions, or requests (at Scragg@cs.Geneseo.edu).

0

Introduction

A journey of a thousand miles must begin with a single step.

CHINESE PROVERB

Pointers1

GatewayBasics.2 This helpful tool can be

Lab: executed at any time.

Lab Unit 0: Getting Started: First Computer

Manual: Access. This unit includes instructions
for getting on a computer for the first
time.

0.1

How do you Solve Problems?

How do you solve problems? Do you attack them systematically or plunge right in? Do you solve a particular problem or do you attempt to find a general solution that you can use for additional problems? If you use a variety of approaches, how do you determine which one is appropriate for a given problem? How is it that some people seem to be able to solve problems quickly while others get mired in the muck? Why can some persons solve some problems very well, but cannot even begin to solve others? It is worth considering these questions before attempting to improve your problem-solving ability. But answering these requires first asking even more basic questions:³

1. In computer science terminology, a *pointer* is a tool for referring to a distant object. In this case the pointers are the names of the related units. Actually, any footnote is a pointer to information out of the mainstream of the text. You will find pointers of this and other forms

throughout the text and meet a closely related term, *address*, in Section 16.3.

2. *Basics* is either *Macintosh Basics* or *Windows Tutorial*, depending on the type of machine you are using. Unlike most Gateways, *Basics* is created by the vendor and supplied with the computer at the time of purchase. These tutorials provide the essential information needed to start using a computer.

3. Notice that the process of answering one question seems to raise two more questions. This will be a common phenomenon throughout this text, and it represents a basic technique in computer science problem solving: *divide and conquer*.

1. What is a problem?

2. What is a solution?

0.1.1

What is a Problem?

My dictionary defines *problem* as

1. *A question or situation that presents uncertainty, perplexity, or difficulty.*

2. *A source of trouble or annoyance.*

3. *An intricate or unsettled question; a source of perplexity or vexation.*

In a problem-solving course, each of these definitions will seem to be the most important at different times. Certainly, the instructor will assign problems to students and students will find solutions. But the real goal of those problems is to prepare students for future problems. Everyone faces intricate and unsettled questions in life, problems that can cause perplexity or vexation. *Problem solving*, as used here, generally refers to generalizable approaches those you can use again and again to deal with future problems. Most of the approaches are relatively free of any specific context, but can be applied in any field.

Box 0.1 What Number Comes “First”?

Computer scientists have an unusual custom: when they count, they start with zero, rather than one. For example, notice the numbers of this chapter and its subsections. The practice is not really as unusual or as arbitrary as it seems. For example, birthdays are always counted from zero. Your zeroth birthday is the day you were born. Your first birthday does not come for another year. All the days of your first year of life come before your first birthday.

In the process of exploring the concepts in this text, you will discover some very good computer science reasons for this apparently aberrant behavior. In the meantime, please indulge us our idiosyncrasies. The use here suggests that this chapter marks the starting point of the text; it comes immediately prior to the usual beginning of the text. *Chapter 0* contains material that describes what the remaining chapters contain. Although related material is often placed in the preface, *PSC* includes it here in the text proper because it provides the first introduction to computer science principles. Similarly, the first section of each chapter (*n.0*) provides an overview or introduction to the remainder of the chapter.

Some problems are made explicit at the time they arise. For example, your boss may come to you and say, “We need a design for a new bridge,” or “We need to increase sales. You find a way.” Unfortunately, most problems assigned by your instructors are also of this form. I say *unfortunately* because most problems in life are not explicitly stated. Rather, they are implicit in any attempt to find a solution for some other goal. For example, your parents may want (request, suggest, urge, require) you to get good grades. However, in all probability, they do not tell you how to get them. Nothing in the problem statement “Go to the store,” contains the problem “Get the car unstuck from a snow bank.” But the latter might be necessary in the process of solving the former.

0.1.2

What is a Solution?

Again, my dictionary provides multiple answers:

1. The method or process of solving a problem.
2. The answer to or disposition of a problem.

Problem solving must address both issues. First you must understand what result you want. Then you must understand how to accomplish that result. For example, if you wish to write a letter to the dean complaining about an instructor, you must know what you want to say (“Professor *X* is a no-good so and so.”). Then you must decide how you want to say it (perhaps by explaining the bad things that the professor has done to you), what action you would like the dean to take (perhaps dismissal), and so on. When dealing with a computer, a third issue emerges:

3. Instructing a computer to accomplish the desired result.

Writing a paper with a word processing system requires that you tell a computer how to record your ideas. How do you convey your specific intentions to a computer? How do you control the way the

paragraph looks? How do you check the spelling? These are neither part of the process of problem solving nor the final answer. *PSC* addresses all three aspects, but the last is really a by-product. Unfortunately, the last is often the most obvious problem addressed in an introductory text.

0.1.3

Recognizing and Evaluating Solutions

So, a problem is something that you desire to solve. Does that mean that every problem has a solution? In general, it is impossible to know if a given problem even has a solution, or if it has a solution, to recognize that solution.⁴ If you know

4. The German-American mathematician Kurt Gödel proved this remarkable assertion more than ten years before the invention of the first electronic computer. See the box in Section 14.4.2 for more on “impossible problems.”

there is a solution and you have an approach to finding it, how do you know when you have actually found the solution? Not only is it difficult to recognize a solution, but a given problem may have many solutions. Sometimes this requires finding the best solution. But even the word *best* may depend on many factors: the individual problem solver (the method for grabbing a box off the top shelf may be different for Shaquille O'Neal than it is for Danny de Vito), the tools available (not everyone has a computer available), and the circumstances (if you know that you will have to address the identical problem again at a later time, you may want to put more effort into the solution the first time; sending a letter to someone on campus may require a different solution than sending a letter home for money).

Unfortunately, examples of nonsolutions are far more prevalent than most people would wish. The Vietnam conflict was not a solution to the perceived problems of the region. Prohibition was not a solution to alcohol problems. And the automobile was not a solution to the problem of pollution of city streets (due to horses). Unfortunately, the failure of these solutions was not perceived until after they were attempted. The task of any problem solver is to find a solution that really is a solution not an exchange of problems. This course helps develop tools and techniques for analyzing solutions.

0.2

Computer Science and Problem Solving

Why should a course that aims to teach non-computer scientists about problem solving base its approach on the methods and approaches of computer science? How can computer science help the casual computer user? The methods of computer science represent tools that have been developed over many years. They can often be expanded to more general forms.

Specifying a Problem.

Understanding a problem is the first, and perhaps most important, step in solving a problem. Computer science has been described informally as the *discipline of problem solving*. So, it should not be too surprising that computer science has developed both a system and a language for describing problems.

Specifying a Solution.

More often, computer science is defined as the study of *algorithms*: a series of instructions that constitute a solution to a problem. The class of problems for which algorithmic solutions exist is roughly the class of problems for which computers can be used to solve the problem.⁵ That is, computer scientists are interested in solutions they can write down and give to a computer. In general, computer scientists are interested in studying the properties of solutions to problems, for example, how fast a given problem can be solved.

5. This is another remarkable observation put forward before computers were invented; this one by Alonzo Church.

Understanding the nature and limits of solutions tells much about how to solve problems in general.

Specifying Instructions.

How do computer scientists study solutions to problems? One way is to study the steps of an algorithm. Specifying instructions to a computer is essentially the same as telling a person how to solve a problem. There are minor differences. For example, computers are not very good at guessing missing or incorrectly specified details. If you are explaining how to get to a shopping mall and you forget to mention getting into the car, the listener can probably fill in the missing details. If you incorrectly describe the cost of a loaf of bread at a given store as \$100, the listener can probably guess that you left out the decimal point and really meant \$1.00. Most computers would have great difficulty filling in such details.

Problem solving using a computer can sometimes feel frustrating due to this apparent lack of understanding or intuition on the part of your “teammate.” Modern systems provide tools (many described in this text) for reducing these misunderstandings. More important, computer science offers a number of tools for ensuring that instructions are specified correctly in the first place.

Recognizing and Evaluating Solutions.

Finally, the field that studies solutions has naturally developed criteria for evaluating solutions. What is a good solution? How do you know when you have found one? Computer science provides tools for the entire problem-solving process.

0.2.1

Some Problem-Solving Methods from Computer Science

The problem-solving methods explored in *PSC* represent the three major problem-solving paradigms of computer science:

analysis

design

modeling

The “official” descriptions of the *Discipline of Computer Science* use the term *model* in the sense that a scientist uses it: an attempt to describe the essence of a phenomenon. There are actually two important subparadigms that computer science has borrowed from the older sciences:

modeling

empirical verification, or testing

Analysis.

Analysis is the separation of a whole into its fundamental elements or constituent parts; a careful, detailed, and often formal study designed to uncover pertinent information. Formal logic and mathematics provide especially rich collections of tools for analyzing problems and solutions. But you need not use

complex mathematics to gain some impressive results. Simple analytic tools include:

approximation: estimating some value (such as size, distance, or composition), without attempting to get an exact answer. This tool is invaluable in judging solutions. Approximation should always be the “first line of defense.”

quantitative prediction: understanding what an answer should look like. Prediction may be one of the most important tools in problem solving. Far too many “solutions” could have been rejected if the problem solver had done some preliminary work to predict the form of the final answer.

problem analysis: finding a clear statement of the problem. This is one of the most neglected aspects of any solution.

Design.

Design refers to the *construction* of a solution, rather than the *finding* of a solution. The problem-solving process is not nearly as random as the latter would imply. Design is the application of a method for making or doing something or attaining a goal.

Programming is the design of algorithms for the solution of problems. Any complex project requires design before implementation. Aspects of the design process include:

development: the progressive advancement from a lower or simpler to a higher or more complex form. In particular, computer scientists are interested in the process of developing general solutions to computing problems.

direction: moving *toward* a goal. One must establish the goal before attempting a solution. The design must pursue the correct direction.⁶

divide and conquer: breaking one large problem into two or more smaller ones. One of the best techniques for attacking large

problems, this depends on the observation that two small problems are easier to deal with than one large one.

Modeling.

The creation of models is an essential tool for all science. Models enable us to understand new concepts and to predict results of new situations. Finding the correct modeleither conceptual or physicalfor a situation is often half of the battle in problem solving. Important modeling techniques include:

abstraction: the distillation of the important or essential concepts from a more complicated whole. Abstraction enables you to find general solutions so that you need not reinvent the wheel every time you commence to solve a new problem. Abstraction itself is sometimes called modeling.

analogy: agreement or correspondence in details (such as appearance, structure, or quality). Analogy is perhaps the best tool for gaining a quick understanding

6. Notice the example of the ambiguity of English. *Direction* also refers to a step or an instruction in the process of creating an algorithm. Disambiguation is yet another goal of computer science methods.

of an apparently new concept. Using a known concept as a model for a new concept helps build understanding of the new tool quickly.

visualization: tools for representation, such as graphics and numerical problem solving.

Empirical Verification.

Empirical means “originating in, relying on, or based on factual information, observation, or direct-sense experience.” Carefully applied empirical methods are often called the *scientific method*, and include:

experimentation: an operation or process carried out to resolve an uncertainty, particularly when conducted in a controlled manner. Every solution requires at the very least a trial run to see if it produces the expected result. More formal experiments are also an important tool.

diagnostics: the use of information about your mistakes to design better solutions. Without diagnostics, successive attempts at solving a problem would not bring you closer to a solution.

evaluation: the act of appraising or valuing the nature, character, quality, status, or worth of something in particular, the results of an experiment or the application of an algorithm.

experience: certainly the best teacher of all. Experience is essential for understanding any concept.

As the course proceeds, these elements reappear repeatedly. These are the tools that form the essence of *Problem Solving with Computers*.

Exercises

Great thoughts reduced to practice become great acts.

WILLIAM HAZLITT

0.1 After reading all of the new words presented thus far in Section 0.2.1, write down as many of the terms as you can remember.

Example 0.1 Computer-ADE *Mnemonics* are popular among computer scientists as an aid to the problem of keeping track of information. Technically, any tool, such as a rhyming pattern, that helps you remember information is a mnemonic.⁷ The previous several pages describe a number of tools and concepts in rapid-fire succession. It would be a very good student indeed who remembered all of them after a single reading. The helpful mnemonic

7. A mnemonic made from the initial letters of several words (such as *pin* for “personal identification number”) is called an *acronym*. Although the tendency of computer scientists to use acronyms heavily seems to contribute to the image of computers as “incomprehensible,” you will see that their proper use can actually make many problems easier to understand.

Computer-ADE is based on the three paradigms: *analysis*, *design*, and *experimentation*. The “ade” in “Computer-ADE” should be thought of as in lemonadea refreshing way to quash your problems and anxieties. It is not computer-aidwhich might mean a tool to help computers or perhaps an illegal scheme to funnel money from the Third World into computers. The mnemonic is simple but useful: almost all of the various techniques start with the three letters A, D, or E:

A: analysis, analogy, algorithm, approximation, abstraction, and acronym

D: development, design, diagnostics, direction, divide and conquer

E: empirical methods, experimentation, evaluation

Exercises

0.2 After reading the mnemonic in Example 0.1, repeat

Exercise 0.1. How many of the terms can you remember now?

0.3 At this point (if you have not already done so), it’s time to get started. Go explore your computer hardware. Complete the *Gateway Lab Basics*.

0.4 *Laboratory Unit 0* of *PSC Lab Manual* contains instructions for starting the process of learning about the hardware you will be using. This includes activities to be completed before your first scheduled lab session. These “prelab” activities are an essential part of the learning process. Look at this lab unit now.

0.3

Summary

Each chapter ends with a very brief summary. Chapter 0 provides

an overview of the intended purpose and use of the remainder of the text. In particular, it should be seen as a selected introduction to computer science”selected” in the sense that it focuses on those aspects of the field that are likely to be of more use to non-computer scientists. Each chapter summary includes a brief summary of the important concepts of that chapter.

Important Ideas

problem solution computer science

application mnemonic graphical user
interface

evaluation experiment recognition

wysiwyg analogy specification

1

Using Tools: Word Processors (A Case Study)

The human body is a magazine of inventions, the patent office, where are the models from which every hint is taken. All the tools and engines on earth are only extensions of its limbs and senses.

RALPH WALDO EMERSON

Pointers

Gateway Basics

Labs: Basic Word Processing1

*Lab Unit 1: Getting Started: Representing
Manual: Text Data Unit 2: Finding Your Way
Around*

1.0

Overview

To most people, the word *tool* refers to hand tools or perhaps power tools. Although Chapter 0 describes computers as tools, many people do not think of them in that way. Thinking of computers as tools is an essential step in learning to use them well. As with any other tool, the first part of that step is understanding how it is used.² Fortunately, any new computer user already knows quite a bit

1. If you have not already done so, I strongly recommend that you complete these *Gateway Labs* at the earliest possible time. The sooner you start them, the easier it will be to understand the text. If you have trouble getting started, see Laboratory Unit 0 for help.

2. *Problem Solving with Computers* parts ways with many introductory texts on this point. The underlying premise here is that understanding the use of any tool is essential; understanding what is inside the machine is

not. When learning to drive, understanding the effect of turning the steering wheel is more important than understanding the physics of the internal combustion engine (see Section 0.2.1).

about using tools. That knowledge is just as applicable to computers as it is to other tools. Chapter 1 uses familiar categories to describe common tools, and then extends those categorizations to computerized tools, illustrating these concepts with *writing tools* throughout.

1.0.1

Is Writing a Problem?

According to the preface, this text is about problem solving. In that case, why focus on writing tools? Writing is indeed a problem.

Many college faculty think that the biggest problem new college students face is poor writing skills. By any of the more formal definitions of *problem* (Section 0.1.1), writing also qualifies.

Writing is the conveying of information. Specific writing problems range from organizing your thoughts and writing convincing arguments to finding the correct spelling and punctuation.

Computerized writing tools are more than a substitute for a typewriter; they can help with virtually every writing problem. No tool will make a problem go away completely; they only make a problem more manageable. Used well, computerized writing tools can truly improve the quality and results of your writing. Used poorly, they can even make it worse.

Just as important, writing tools provide an excellent stepping stone on the path to more specialized problem-solving tools. Every student understands the basic actions of writing. It is only a short step to understanding the basics of writing tools. From there, it is another relatively short step to understanding more sophisticated tools.³

1.1

Describing Tools

What is a tool?

A tool is a device or instrument used to perform work.

A tool is anything used to perform an operation or task.

Thus, hammers, chisels, brooms, erasers, shovels, compasses, woks, coffee makers, sewing machines, automobiles, dictionaries, crib sheets, rulers, printing presses, fingernail cutters, wrenches and computers are all tools.

Discussion requires a vocabulary. Discussion of tools requires words for referring to specific tools, groups of tools, and properties of tools. Individual tools can be classified in any of several ways. The most common classification of tools is probably by *domain* of use: the general areas of application for the tool. Common domains include gardening tools, mechanic's tools, cooking tools (utensils), and so on. Perhaps domain could also be called "occupation of user": gardener, mechanic, cook, and so on. A computer is a very general tool; no single

3. Do you recognize this as the *divide and conquer* approach from Section 0.2.1?

domain seems to describe its use. There are word-processing tools, financial tools, mathematical tools, drawing tools, and others. For understanding computerized tools, classifications other than domain seem more useful. Two good classification tools for describing tools in general are *flexibility* or *generality* (for example, a Swiss Army Knife versus a paring knife), and *power level* (for example, hand versus power tools). Other possibilities include the type of objects the tool normally operates on (such as food-preparation tools or woodworking tools), the types of operations they perform (such as cutting (saw and butcher knife) versus measuring (tape measure or measuring cup)), and their intended application area (scissors and X-acto knives both cut paper but have different applications). Table 1.1 illustrates several classifications and examples.⁴

1.1.1 *Flexibility*

A computer is a tool, but a more realistic analogy might be to think of a computer as a tool box, or a collection of many tools for solving a wide variety of problems. A cook has many tools: beaters, bowls, pans, stove. So does the carpenter: saw, hammer, plane, level. For the computer user, a computer becomes a sort of multipurpose tool, like a *Swiss Army Knife*. Just as the knife may have a knife-blade, a screwdriver-blade, scissors, a toothpick, and any of several other “blades,” a computer can serve as any of several tools. It can be a *word processor* for manipulating words and text, a *spreadsheet* for manipulating tables of numbers, a *communication device* for accessing distant sources of information, or a *database* for organizing information. It all depends on the “blade”: the *application* or specific *software* tool. A word processor is just one blade in the knife. Surprisingly, a word processor itself is actually a family of tools (or

TABLE 1.1 Tool Classification

Major Classification	Example of Classification	Example in the Class	Example of Classification	Example in the Class
Function Objects	Cutting Food	Knife Fork	Connecting Freight	Hammer Fork lift
Power level	Hand	Hand drill	Power	Electric drill
Flexibility	General	Automobile	Specialized	Race car
Granularity	Small	Pencil	Large	Postage stamp

4. Table 1.1 could be described as a “classification of classifications.” Definitions involving *self-reference* are part of a recurring theme in computer sciencea surprisingly useful theme. In this case, the notion of classification is applied to the notion of classification. Watch for future instances of this strange theme, for example, Footnote 1 in Section 2.1.1.

subtools) like the Swiss Army Knife, or the entire computer. Both a word processor and an entire computer are tools with different levels of flexibility.

Exercises

- 1.1 Table 1.1 does not include the classification “domain” described earlier in this section. List five domains not discussed in the text and give two examples of tools for each of those domains.
- 1.2 List three tools like the *Swiss Army Knife* that have greater-than-average flexibility.

1.1.2

Use or Function

A knife is a tool for cutting objects; a needle, for sewing; a fork, for eating; and a pen, for writing. A computer is a tool for problem solving. A hand-calculator and an abacus are also tools for solving arithmetic problems. A dictionary and a thesaurus are tools for solving language problems. Computers can be used as tools for solving many kinds of problems; each particular subtool, or *application*, helps solve a specific subclass of problems.

Functions of Writing Tools.

You are already familiar with many tools for the task of writing. The primary tool is usually a writing instrument anything capable of putting characters on a page. Before the introduction of computers, these tools included typewriters, pencils, pens, chalk, and crayons. A *word processor* is a computerized tool to replace manual writing tools such as a typewriter. But the writing activity, taken as a whole, requires more tools than just a typewriter or word processor. The letter must be written on something: paper or a clay tablet. Humans make mistakes, so we also need tools for correcting: erasers,

backspace keys, and so on. We need to deliver the written objectterm paper, job application, or love letterto someone who will read it. The U.S. Postal Service, Federal Express, and campus mail are all tools for delivering the letter. We need to store the letters somewhere. File cabinets and shoe boxes both fill this function. Sometimes we want to include material from other sources in our documents. Photocopiers and scissors can both help with that problem. Each of these groups of tools has an electronic *analog*a tool that performs the same function. A typical word processor can help write a *document*, modify it, fix the spelling, and change its appearance.

5.*Subtools* represent another common computer science theme related to the self-reference theme of Footnote 4. The computer is a collection of tools, and each of these tools is a collection of tools; an item is made up of subparts, each similar to the original item. Similarly, notice the use of parentheses nested within parentheses such as “cutting” in the last sentence before Section 1.1.1.

1.1.3

Power Level

A tool is but the extension of a man's hand, and a machine is but a complex tool. And he that invents a machine augments the power of a man and the well-being of mankind.

HENRY WARD BEECHER

You have certainly used both hand tools and power tools. Each can be found in any domain. A power tool usually has a hand tool analog tool that does the same things, but without an external power source: hand saws versus power saws, hand egg beaters versus electric egg beaters, bicycles versus motorcycles. A power tool has the same purpose as a hand tool it just makes a task easier, faster, or more accurate.

People use tools to manipulate objects. A saw is used to manipulate wood; a car to transport people or cargo. The tool does not perform any actions on its own. Ultimately, the human *user* of the tool is responsible for the use of the tool, causing the tool to perform its function and guiding the overall process. The human makes the decisions and controls the tool. In the case of hand tools, the human manipulates the object almost directly. The human physically moves the saw up and down or cranks the egg beater. With a power tool, the human gives a *command* to manipulate an object, say by pushing the button labeled "perform this action." With a writing tool, the human must create the written document. A computerized writing tool is a power tool that makes the task easier. Anyone can add numbers by hand, but it is faster to use a spreadsheet; a library card catalog is useful, but a computerized version is more versatile.

Exercises

1.3 List as many writing tools as you can. Describe each in terms of its subfunction and its flexibility.

1.4 With another student, try this game. One person names a hand

tool. The other person must name a power tool analog for the hand tool. Score one point for each hand tool for which the second person can name no analog. Try it in reverse: name a power tool and attempt to name the hand tool.

Limits of Tools.

One machine can do the work of fifty ordinary men. No machine can do the work of one extraordinary man.

ELBERT HUBBARD

Neither the hand tool nor the power tool can solve a problem that a human cannot. They do, however, reduce problems from *intractable*⁶ to merely difficult, and from

6.Intractable is a formal term in computer science used to refer to problems that are so hard to solve, we might as well not try. Here, it has its more everyday definition: “extremely difficult to manage.”

difficult to reasonable. You can cut a piece of wood without a knife by rubbing a piece of broken glass against it, but it is tough on the fingers, and not very neat. You can cut 1000 boards without a power saw, but it is certainly slower. Similarly, computers are tools for solving problems faster, more neatly, more accurately, and cheaper than by hand. A word processor makes it easier to produce a quality written document. It does not perform any conceptual work, but it does greatly reduce the drudgery, the errors, and the time.

1.1.4

Objects, Tools, and Users

Tools manipulate *objects*. Hammers pound nails, saws cut wood, and forks lift food. Then what is the object of a word processor? If a computer or perhaps the computer together with the word processor is the tool, what are the objects? One is obvious: paper, or its electronic equivalent, the *document*. We could also say that the pen and typewriter manipulate text, placing letters on the paper. Formally, we refer to the thing being manipulated as the *object* or the *operand*. Each action or *operation* writing, erasing, and so on changes the object. The tool does perform the operation but at the request of a human. The human uses a pocketknife or a power saw to cut; it is the tool that does the actual cutting. The human controls the tool and instructs the tool how to perform the operation. We call the human the *user*, and the tool, the *operator* or *operation*. Entering a character, cutting a segment, and printing a document are all operations. The use of *operator* to refer to an action rather than the user comes from the mathematical use of that term to refer to mathematical or arithmetic tools, such as the addition operator for calculating sums, or the multiplication operator for calculating products.

1.1.5

Granularity

The scale of the actions a tool performs provides a classification most people do not think of immediately. The driver of an automobile has access to many controls. Some of these control the entire car: the brake, the accelerator, and the steering wheel. Others control only individual parts of the car: the turn indicators, radio controls, and power window switch. Writing tools have at least two granularity levels:

those used to create individual pieces of text (type a character, erase a word)

those that manipulate an entire written document (postal delivery service, file cabinet, duplicating machine, Federal Express)

The first set actually manipulates only a small part of a larger object at a time: one word or even one character at a time. The latter manipulates the complete document. We call this difference *granularity*. Most writing tools have a low or small *granularity*: the principal objects (letters and words) are small compared to the completed full letter. Other tools, such as electronic mail, have higher granularity; they are designed to manipulate the entire document as a single object. The user needs to select a tool with the appropriate granularity for the task.

Exercises

1.5 Classify cooking and woodworking tools by large or small granularity.

1.6 Contrast the word *operator* as used above with its common English language usage.

1.2

A Word Processor as a Tool

The basic functions of a *word processor* (operator) are exactly the same as their traditional typewriter counterparts. In fact, in many ways, using a typewriter and using a word processor are more alike than using a pen and using a typewriter. The user types at the keyboard and new characters (objects) appear in the document. Typing the first five letters of the alphabet into a blank line:

Box 1.1 Selecting Tools

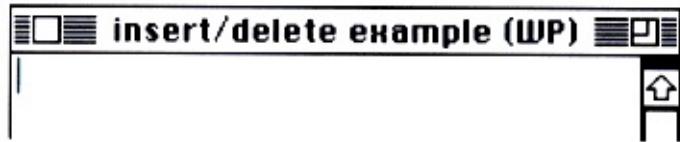
Are All Word Processors Created Equal?

The basic tools in every word processor are essentially the same. Not only are the basic capabilities identical, but the mechanisms for controlling the tools are, if not identical, equivalent. Only the bells and whistles vary. Common word processors include

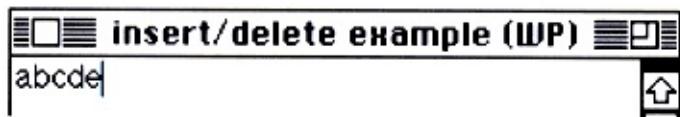
Microsoft Word®, *WordPerfect®*, *WriteNow®*, and *MacWrite®*, as well as the word-processing tools which are part of larger integrated tools such as *Microsoft Works®*, *ClarisWorks®*, or *Great Works®*. The general concepts covered here are applicable to all of these specific word processors. Significant differences within word processors exist primarily at advanced levels or for very specialized features. For example, not all word processors support boxes such as this one.

Are All Computer Interfaces the Same?

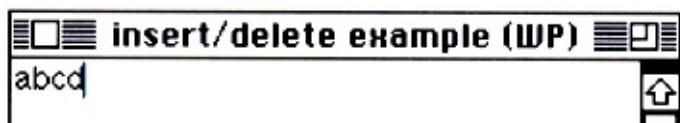
Although *Problem Solving with Computers* describes a generic picture of computing, dependent on no specific computing system, it does assume that you have a modern interface. For the majority of students this implies that they are using either a *Macintosh* or a machine running some version of *Microsoft Windows®*. Other possible gui interfaces include the *OS/2®*, *X-windows©*, and *GeoWorks®* systems, and those used by workstation computers such as *Sun SparcStations®*.



produces



To erase the last character, the user *deletes* or *backspaces*,



and then



and so on. Essentially it is a typewriter; more powerful because it is easier to use and more versatile because, like the Swiss Army Knife, it contains all the blades in one place.

Exercises

*Learning computer science by reading about it is like making love by mail.
(WITH APOLOGIES TO) LUCIANO PAVAROTTI⁷*

1.7 Use a word processor to write a sentence stating why you are taking this course. Delete the last several words and replace them with a new reason.⁸

7. Pavarotti was actually talking about music not computer science but the point is just as valid.

8. Yes, you are being asked to jump in and try this with little help. But help is available: Unit 0 of the *Lab Manual* describes the details of how to do this. Also, the two *Gateway Labs* pointed to at the beginning of this

chapter will help you get started.

If possible, try to have a computer handy as you read these sections (and indeed many or even most sections of the book) to help you explore new ideas as soon as they appear. Do not attempt to just read and memorize the ideas. It is not a good learning technique.

1.2.1

Selecting Objects

When using any tool, the tool user must select an object to work on or manipulate. Many hand tool actions implicitly select a physical object. To cut wood, you must first select the piece to cut. To beat eggs, you must first select the eggs to beat. An explicit description of an action needed for telling a computer or another human how to perform the action requires both the tool or action (operator) and the material (object): cut the paper with the scissors, cut the steak with a knife, cut the wood with a saw. Write a letter with a pencil, write a check with a pen, write your term paper with a typewriter (or better yet, a word processor). No English description of an action is complete without specifying both the operation and the operand. Similarly, a computer user must always specify both action and object.

Selecting Text Objects.

A typist selects a character by selecting a key. The location of the character is selected by default: it is the current location of the carriage or typeball. Generally, that is immediately after the last character typed. Typing anywhere else requires deliberately moving the typehead (or *platen* or roller bar). Similarly, using a word processor, the new text is generally entered immediately after the last character previously typed.⁹ The user changes the location of the next character by changing the *insertion point* usually a flashing *cursor*, which may take any of several forms:

| or] or a

The action “enter a new character” requires an operation (typing) and operand or location (insertion point). Always select an insertion point before attempting to insert characters at a new location. A description missing either part does not specify a complete action. Consider the difference between the two scenarios depicted in

Figure 1.1 on page 18. In the first case, deleting accomplishes the objective described by the example. The second case creates a very different result.

Default Operand.

The *default* location for word-processor operations is the last location previously typed.¹⁰

Data as Objects.

Another kind of object is less obvious: the *data* or *informa-*

9. A conundrum: If the new character always goes after the previous character, where does the first character go? (See Section 5.5.4.)
10. Generally, this text does not concern itself with defaults. But default location is unusually important. The ability of a word processor to pick the proper default location obscures the need to specify the operand.

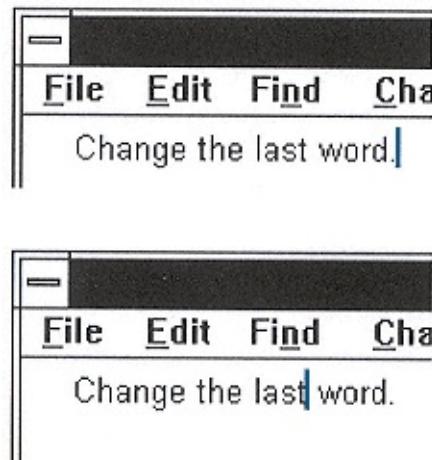


Figure 1.1
Two Different Delete Actions

*tion,*¹¹ in the case of a word processor, the text or the numbers manipulated by the tool. Data and information are abstract rather than concrete objects. Although information is intangible, it is the most important of all objects. Computers are often called *information processors*, reflecting the fact that their power lies in their ability to manipulate data and information. Abstract objects are discussed further in Section 2.3.2. For now, think of the characters and paper as the objects.

Exercises

1.8 Use the sentence you created in Exercise 1.7 as a starting point.

Select any three (nonadjacent) words. Change them, without changing or retyping any other words.

1.9 It is time to use a computer. If you have not already done so, go now and run the *Gateway Labs*:

Basics, sections 1-3, and
Basic Word Processing

Seriously! Stop reading and go do the labs.

11. Technically, *data* refers to raw input and *information*, to the results

of processing the raw input. Many people insist on this distinction, but we will not quibble about these differences here.

1.3

Using Descriptions to Understand Actions

The function of a tool may be obvious from its description, and its flexibility may become obvious with use, but descriptions of other characteristics help clarify how the tool is used.

1.3.1

Granularity

Most complex tools perform actions at more than one granularity, but work best at some specific granularity. A hand shovel works best as a garden tool or for digging small holes. It can also be used to pick lint off a carpet or to dig a basement, but other tools are better suited for those tasks. Similarly, word processors have capabilities for handling entire documents, and electronic mail applications allow users to manipulate single characters but each is best at its own granularity. For example, in addition to manipulations of individual characters, word processors can perform the following operations on an entire document:

Print an entire document: create a paper copy of the document that previously existed only in electronic form

Save an entire document: make an electronic copy for future use

Open an entire previously existing document

Close it when you are done

Create a New document

However, the real strength of word processors is in manipulating a few characters at a time. Most word processors, and indeed, most computer applications, support higher granularity operations as well as specialized lower-granularity (e.g., character-oriented) operations. Similarly, applications that deal primarily with higher granularity activities usually provide some (perhaps primitive)

lower-level capabilities.

Most applications organize their command menus according to granularity levels: commands that manipulate the entire document or file in one menu (e.g., File) and those that manipulate only a small segment in another (e.g., Edit). Other menus may be grouped by the function of tools (e.g., graphical tools), or by type of object (e.g., windows).

Exercises

Note: These exercises all involve high-granularity actions, which are usually all grouped together in a single menu named something like File.

1.10 Create a New document. Type your name and address. (If your word processor automatically opens a new document on start up, skip this exercise but be sure to do Exercise 1.15.)

1.11 Save the document you created in Exercise 1.10.

(table continued on next page)

(table continued from previous page)

- 1.12 Print the document you created in Exercise 1.10.
- 1.13 Close the document you created in Exercise 1.10.
- 1.14 Open the document you just closed in Exercise 1.13.
- 1.15 Create another New document. Type the basic information about this course: course name and number, instructor, time, and room number.
- 1.16 An experiment: Most word processors protect you against accidentally throwing your document away. If you attempt to close an unsaved document, the word processor asks whether you would like to save it. Does yours?¹²

Creation Versus Delivery.

Individual actions needed for creating an object tend to have smaller granularity than the actions that do something with that object after it is created. While creating a letter, the writer manipulates individual letters. Once a document is complete, the owner can do things with it: mail it, post it on a bulletin board, take it somewhere by hand, make a Xerox® copy of it, or even throw it away. These actions are performed on the entire message (i.e., they have large granularity). Any given action using the typewriter, on the other hand, acted only on single characters of the message (i.e., with small granularity). Selecting a tool with the correct granularity is the first step in using the tool to solve a problem. A word processor is a tool for creating a document. Therefore, most of its individual subtools have a small (e.g., one character) granularity. It is not a particularly good tool for delivering a letter.

In contrast, delivering a letter requires a very different tool. The traditional paper world includes several tools for delivering a letter

(e.g., post office or courier). A whole new set of tools is available for delivering electronically created letters. *Chapter 3: Documents as Objects* discusses tools for manipulating entire written documents, including filing documents and sending letters through electronic mail. This classification of actions corresponds to two views of a text document that differ in their granularity. The document is composed of characters for which a text processor may be the appropriate tool. On the other hand, a mail delivery tool would be good for getting the letter delivered. Three especially important ones are discussed later in the next chapter.

1.3.2

Intermediate-Level Objects

Groups of Characters.

An object may fall in between the two large and small granularities described above. For example, a group of several charactersa word,

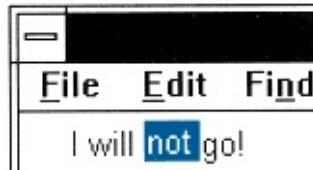
12. This experiment is really not very dangerous. The worst thing that could happen is that you could lose what you have typed . . . and it is far better to have that happen now in a controlled experiment than later in the middle of your term paper, right?

a sentence, a paragraph, or some more arbitrary grouping can be an object. Manipulation of groups of characters creates some new questions:

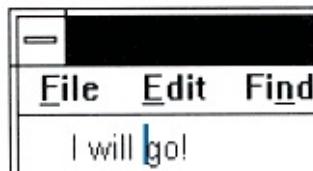
How do you specify a group rather than a point?

What does it mean to insert a character at the selected “insertion point” if the location is not a point but a group or range of characters?

For example, given the text:



with the word “not” selected. Deleting the selection removes the entire selection not just the last character of the selection:



Combining Operations.

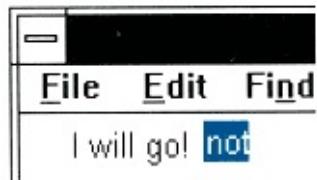
Just as deleting a block of selected text removes the entire block, typing text with a block selected replaces the entire block with new text. (See Section 5.5.4 for a discussion of the consistency of this.)

Is there a way to recover characters that have been deleted? Yes, but it requires a slight modification of the delete concept: *cut*, defined as

Delete, but don't throw away.

Cut objects can be retrieved later and *pasted* at a new location. Suppose that in the above example, the word “not” had been *cut* rather than *deleted*. If the user selected a new insertion point at

the end of the sentence, pasting the word back would yield:



Cut and paste is the first concept covered here that is not supported by a common typewriter! However, it is an essential tool for reorganizing a written document, a tool that gives true power to its user. You may move any misplaced block of text to

any other part of the document. This means that a writer can rethink the organization of a document at any level, not just a character at a time, moving words, sentences, paragraphs, or other segments. Working and thinking at different conceptual levels is essential for any intellectual task.

1.4

Summary

A computer is a multipurpose tool for manipulating and holding information. The user directs the computer to manipulate the information. More important, the user can use abstract tools for understanding how computer applications work.

Actions on Characters (Small Granularity)

insert characters into text

change a text segment

delete a text segment

move (cut and paste) a text segment

Actions on Entire Documents or Sets of Data (Large Granularity)

save

print

new (create a new document)

open (an existing document)

close a document

Important Ideas

document experimentgranularity

applicationanalogy operand

save operator object

tool function machine
tool

selection

2

Tools and Computer Science

We know nothing of what will happen in future, but by the analogy of experience.

ABRAHAM LINCOLN

Pointers

Gateway Basic Word

Lab: Processing

Lab Unit 2: Finding Your Way

Manual: Around Unit 20: Crisis Management

2.0

Overview

Computer science, the science of problem solving, offers several tools for beginning the process of solving problems. This chapter describes several of those tools, which are applicable to both computing problems and problems that do not involve computers in any obvious way. Develop the habit of applying these tools to problems throughout your life.

2.1

The Three most Important Computer Tools

You have already been asked to jump in and start using new tools: computers and word processors. This may be a little scary at first. But every participant in any activity must start somewhere. For neophytes, everything is new all at once. And even advanced practitioners continually find new things to do and new ways to do them. For this reason, virtually every computer application provides

three sets of very important tools:

help,

undo, and

save

For *learning by doing*, these commands are almost indispensable.

2.1.1

Help

You cannot teach a man anything; you can only help him find it within himself.

GALILEO

Almost all computer systems and individual applications now have built-in *help facilities*, designed to help the user use the tool. Help comes both in the form of printed material: manuals, books, and short "cheat-sheets," and in so-called *online* or *interactive help*, provided directly by the computing system, such as the *Windows* Help facility shown in Figure 2.1. In a *learning-by-doing* context, online help is clearly the most important. Online help comes in at least three flavors: *context-sensitive help*, *application-specific help*, and *tutorials*.

Context-Sensitive Help.

The most helpful help would always provide exactly the information needed at a particular moment. Context-sensitive help attempts to do just that. The information provided when a user asks for help always reflects the current computational situation. For example, with Apple's *balloon help* (shown in Figure 2.2), the user can point to an item and a balloon appears containing information about that item. Notice from the figure that *Help*

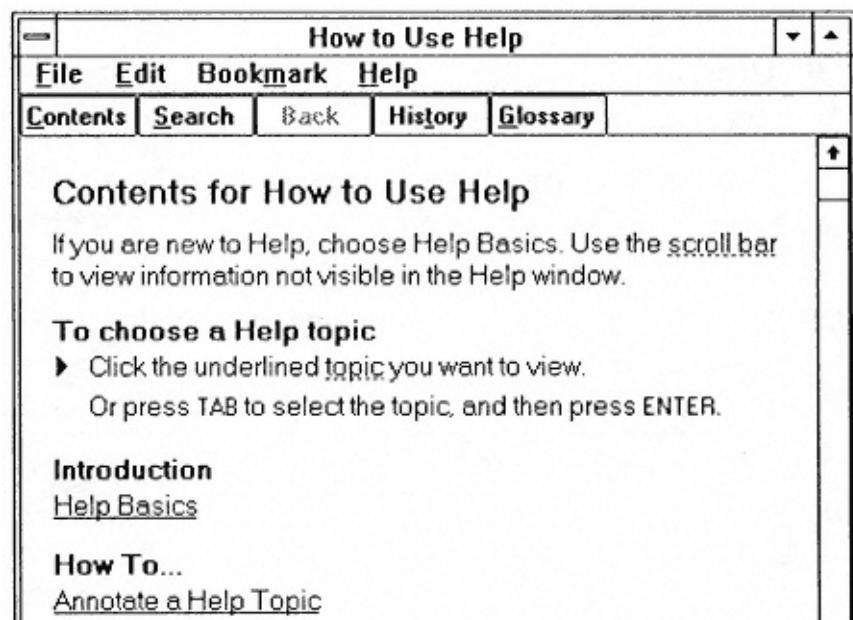


Figure 2.1
The *Windows* Help Menu



Figure 2.2
Balloon Help

commands can even provide help about how to get help!¹ You can use balloon help to explore most applications and the tools on the desktop. This is also a good way to explore when you are “killing time.” Just point at an unfamiliar item and a short description appears. Other forms of context-sensitive help keep track of what the user is doing. When the user asks for help, the system delivers information relevant to current activities.

Application-Specific or Menu Help.

Many applications provide a general question-and-answer help facility. The user may ask questions about any aspect of an application. The help facility may be organized alphabetically (like an index), by subject (like a table of contents), or by some other organization. Figure 2.3 on page 26 shows the help facility for the application *ClarisWorks*. The user in this case apparently chose to bypass the subject area, and is pointing to the index button.

So what do you do if you don’t know where the help is? Hint: there is no penalty for opening a menu and looking. In fact, that might be an interesting way to find out about any application or to generate questions to ask in lecture. (See Section 2.2: And the Most Important Self-Help Tool: Try It!)

Tutorials.

Tutorials are automated lessons in the form of computer

applications that introduce important concepts or techniques. Many manufacturers routinely provide such tutorials with their computer products. The *Basics* tutorial or the *Gateway Labs* are both samples of tutorial-type help. Although they often serve as

1. Self-reference (e.g., “help me with help”) seems to appear everywhere. Compare such self-reference to the logical conundrum described in the box in Section 14.4.1. Also, compare “help with help” to the bootstrap concept described in the box near the end of Unit 0 of the lab manual.

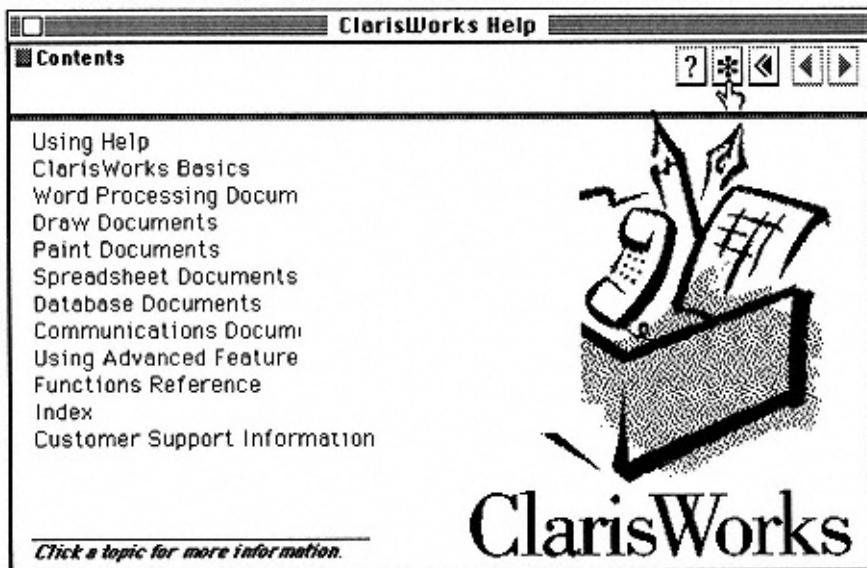


Figure 2.3
Using Application-Specific Help

excellent introductions, tutorials take more control over the lesson than do regular help programs and are therefore not very suitable for use while actually using a tool.

With the help command you almost never need to say, “I don’t know how.” You can always use the tool to help find out how.

A Small Problem with Help Facilities.

Unfortunately most help seems to be organized in exactly the reverse direction from what is really needed. In order to ask about a command, the user must first know the name of the command and then the system can explain how it works. Unfortunately, that is often exactly what the user does not know. The most common form of question is, “How do I do xyz?” which is roughly, “What is the name of the command to accomplish xyz?” The user needs information about which command to use. Thus, using Help requires guessing the name of the command and then looking for a command with that name. There is no penalty for this, but it can be inconvenient.

Fortunately, most applications within a given computer

environment (e.g., *Mac* or *Windows*) use the same name for similar commands. It pays to remember the names of commands used in other applications. Use analogies between the commands to help you remember the names. Also, think of the analogous actions in the noncomputerized world. Finally, it pays to think about the nature of the

command. Use your knowledge of the command to avoid searching irrelevant menus. Print is a large-granularity action performed on an entire document. There is no reason to look for that command in a low-granularity menu such as Edit.

2.1.2

Undo

Once the toothpaste is out of the tube, it's going to be very tough to get it back in!

H.R. HALDEMAN

When a writer makes a mistake with a pencil, a common eraser can remove the error. One could say that the eraser “un-writes” or “undoes” the mistake. It is a little tougher with ink, and a typewriter may be hard or easy depending on whether or not it has an auto-erase feature. In a word-processing system, the user can *undo* the last character typed by hitting the Delete key. Most computer applications provide a much more powerful command, Undo, that “undoes” the most recent actionwhatever it was. Undo is remarkably powerful. Very few commands cannot be undone. Undo feels like a “miracle cure” when you accidentally delete an important paragraph.

The existence of undo means that a wild guess need not cause a problem. If you think that a given command *might* do what you want, *try* it. If it does the wrong thing, *undo* it.

Dialogs.

A variation on Undo is the *dialog*. Many commands provide a chance to think about actions that have potentially catastrophic results. In particular, most irrevocable actions provide an opportunity to retract the command before it is executed. Such commands present a *dialog box*, giving the user optionsincluding *cancelling* the entire request. The dialog may be as simple as “Do

you really want to do that?” For example, the `Quit` command may generate a dialog asking if any open documents should be saved before quitting.

`Revert`.

Another variation on `undo` is `Revert`. Many applications include a command to throw out the entire document and *revert* to the most recently saved version. For example, suppose a student “improved” a term paper by rewriting an entire paragraph, and then realized that the improvements were based on an incorrect interpretation of the assignment. Use `Revert` to return to the previous version.

Beware: `Revert` throws out the good with the bad. Use it only when the problems outweigh any productive changes you have made.

One Caveat:

Although some newer applications allow multiple `Undos`, most do not. This means that only the most recent command can be undone. Check the results of unfamiliar commands before proceeding any further.

Exercises

2.1 Describe² the Undo operation that might or might not exist for each of the following tools:

hammer stapler

saw button

egg beater zipper

2.2 If you have not done so already, go experiment with Undo .

2.1.3

Save

Preserving your investment is imperative. No homeowner would risk such a major investment by forgoing homeowners' insurance. The same precaution applies to art and scholarly research.

Fortunately, this precaution is made easier for computerized documents. Every computer application provides a command for saving the document. Unfortunately, many users do not take adequate advantage of this command. *Always do so.* No matter what Save! Save! Save! Almost any computing catastrophe can be undone by reverting to a saved version.

Save fills two roles. Save records a document onto a computer disk. Until the user says to save a document, no permanent copy (not even an original) exists. Until a document is saved, it exists only in the computer's internal or *main memory*. When the user shuts down the computer, quits the application, or closes a document, the displayed version of the document disappears. The version held in memory or displayed on the screen is temporary. Its life ends when the user quits the application or suffers a power failure. The user must save a document to create a permanent copy on a disk. Fortunately (virtually) all applications remind the user of

this before quitting. But save all documents long before that. Power does fail, and even a momentary glitch can destroy the temporary copy.

Backup.

An important subclass of `save` actions is `backup`. A backup is an extra or reserve copy. A sports team must have a backup plan to use if the original one does not work. Computer users should always keep a backup copy in case something happens to the original. This does not mean just moving the document, but copying it, creating two separate but identical versions. It is best to keep the copies in different places. If you lose an entire disk, having two copies on that disk does not help much. This is very important. If your dog eats the disk, your roommate spills Coke on it, you leave it on a hot radiator, drop it in the snow, just plain lose it, or the computer somehow destroys it, you can simply use the backup copy (see Box 2.2).

2. Obviously, for any assignment that asks you to “write,” “describe,” or “list,” you should use the assignment as an opportunity to practice using the computerized tools. Your instructor will want a neat, well-written, word-processed answer.

Box 2.1 What's in a Name?

On what line? The term *online*, as in *online help*, is not particularly accurate today. Historically, most computing was performed on large centralized computers. Many individual users were connected to a single computer by a cable or *line*. A user connected to a distant machine, is said to be *online*. Online tools then, are tools that are provided by a host computer to the users who are online. The term *interactive help* is more accurate than *online help* for most situations today. But history is a major force, and *online* continues to be the more common description. Such historical anachronisms are unfortunately all too common.

Main? The main memory is also called *internal memory*. This is the primary memory of the computing system. The system uses the internal memory as “working memory.” This memory holds the program and all of your data while the program is working. Main memory is also called *random access memory* or *RAM*. This strange name does not refer to any random event. It refers to the fact that an application can access (almost) any location that it needs (not just the “next” location). The mechanism is not important here, but the name RAM is used commonly in computing literature. (For a more complete description, see Section 3.2, Understanding Memory.)

Find a balance. No one will actually back up at every step of the process. But try to do it often enough so that you can rebuild the document quickly. Think of it as a trade-off. Backing up takes time. If you experience no problems with the original copy, backing up frequently will seem like wasted time. On the other hand, if you have not backed up often and something does go wrong, you will deeply regret the lack of a recent backup. Backing up takes a few seconds; rebuilding an entire term paper can take days. It is very cheap insurance, like using a seat belt. “I didn’t have time to back

up,” is a classic example of the cliché “Penny wise and pound foolish.” In general, back up often enough so that you can easily reconstruct the work you have done since the last backup. For example, computer centers typically back up their entire file system (which may be millions of documents) every day. In the event of a catastrophic failure, no one is out more than one day’s work. Individual users can obviously save their own work more often. Never fall victim to the soothing “It can’t happen to me!” because it can!

Box 2.2 But It Can't Happen to Me!

“It can’t happen to me!” is a common but unfortunate human instinct. It seems that every person who ever uses a computer eventually falls victim to this erroneous assumption. Few people learn to back up regularly the easy way. They do not learn until it *happens*. The “school of hard knocks” is a merciless, but effective, teacher. Almost everyone backs up regularly after the first time they lose a large chunk of data and very few students do so until then. This neglect is the largest single source of problems that humans prefer to describe as: “The computer destroyed my data.” Do not blame the computer. Losing data is only a problem for data that is not backed up. And, for the record, it has even happened to me. More than once. I have spent many a long night making up for my lack of a backup. Backing up is now a well-instilled habit.

And I've Heard Them All Before!

Every one of the problems described in this section has happened to at least one of my students. In virtually every such case, the instructor’s response is the same: “Where is your backup?” Data-recovery tools *may* be able to help you recover lost data, but, by far, the best defense is a good backup. “I lost my data!” in any of the above forms (or even more exotic ones) is not repeat not an accepted excuse for missing homework or labs.

Think about these three tools: Help, Undo, and Save. They are essential capabilities that make computers more powerful than the corresponding manual tools. No carpenter’s saw can undo or save a backup. No hand tool comes with an online instruction manual.

Exercises

2.3 Find out if the word processor you used for the exercises in the

previous chapter provides built-in help . Use it to learn more about finding Help .

2.4 See if your word processor supports Undo . Test the limits of the Undo : can it Undo more than one typed character? More than one line? What about text recently added in more than one location? Can it Undo a Save?

(table continued on next page)

(table continued from previous page)

2.5 Try using backup . Try to save a new copy with a new name.
Try to save a copy on another disk.

2.2

And the most Important Self-Help Tool:
Try it!

*There is no use whatever trying to help people who do not help themselves.
You cannot push anyone up a ladder unless he be willing to climb himself.*
ANDREW CARNEGIE

One essential computer science tool will not be found in any menu: practice or experience. No other learning device is as powerful as experience. If you are not sure that an action will have the result you need *try it*. With the important tools, `save` and `undo`, at your disposal, you should feel completely free to explore. If you make a mistake, you can always recover your original data. For major explorations such as an action that might alter your entire document, or one involving many steps, you may prefer to create a practice scenario. Build a dummy document like the real one (perhaps use `Save as`). Practice on the dummy. When you are confident that you understand how to address the problem, finish the real task. Experimenting with new commands is a much better way to learn about them than just reading about them (ask Pavarotti see the quote before Exercise 1.7). You are more likely to remember what a command does once you have tried it. This is especially true when the first attempt fails.

2.2.1

The Scientific Method

The admonition: *Try it!* represents the simplest form of one of the most basic computer science problem-solving skills: the

experiment. Future chapters discuss a rigorous approach to experimentation, known as the *scientific method*. For the moment, the essential aspect is the attempt itself. For example, suppose you wonder what the command *Open* does. Perhaps you think it will “open” a new paragraph, or perhaps you are just curious. Try it and see what happens; you can always undo the results.

Exploration works best if you approach it methodically. Start by thinking about what you want to happen—not necessarily in computer terms, but in well-formed English. Look for commands that seem related. When you find one that seems like it might do the task, try it. Think about what you expect to happen. Then compare the actual results with your expected results. Without expectations, you may not be able to tell if some subtle commands did anything at all. For example, imagine typing a single character on a partially completed page, to see what happens without thinking your expectations through. Unless you have an expectation that a character will appear at a specific spot, you are not likely to even see the new character. When an experiment performs as predicted, take the time to check exactly what you did. If you have recently attempted several commands, it is easy to forget which one actually worked. If the experiment

“fails,” note what did happen. That information will be useful in figuring out what you should do.

Exercises

2.6 Have you figured out yet how to “get out of” or stop an application on your machine? Experiment to find out how.

Hint: see Exercise 2.3.

2.7 Notice that nowhere in the preceding discussion are there any instructions for shutting a computer down. Many machines provide an instruction that shuts the machine down (rather than just turning the power off by hand). Create and perform an experiment to see if your machine has such an instruction.

2.3

Metadiscussion:

Thinking About the Previous Discussion

Nothing is a waste of time if you use the experience wisely.

AUGUSTE RODIN

The last two chapters contain several descriptions of tools. Those descriptions are also toolstools for understanding and describing tools. This section tries to go one step further by discussing *metatools*, or tools for describing tools.

Box 2.3 Metterminology

The prefix *meta* means “beyond, transcending, or more comprehensive.” It is used throughout mathematics, computer science, and linguistics. For example, a metalanguage is a language for talking about language; metamathematics is the mathematics needed for describing mathematics. In the current situation, the discussion describes the previous discussion. Notice how even this early in the text, a few themes are reappearing: compare the term *meta* to the *self-reference* problem discussed in

Footnote 4 in Section 1.1. A metalanguage makes it possible to talk about a subject within the language of that subject. All of the footnotes on self-reference have been leading to the need for such metalanguages.

For an irreverent discussion of names and meta-names, see the discussion of the song called *Ways and Means*, and whose name is called *Haddock's Eyes*, in Lewis Carroll's *Through the Looking Glass*. Or, consider the famous Abbot and Costello *Who's on First* dialog. In both of these cases, the language for talking about a thing gets confused with the thing itself.

2.3.1 *Analogies*

Many of the terms used by computer users are direct analogies to real-world counterparts. For example, the word-processing terms Cut, Paste, Delete, Copy, Open, and Close all represent analogies to traditional paper-based activities and objects. Recognizing these analogies provides the quickest path to understanding and remembering the terminology and actions. That understanding will, in turn, make it easier to understand more about the tools and to understand new tools when you encounter them. Analogy is a fundamental problem-solving tool.

2.3.2 *Abstraction*

In common usage, *tool* usually refers to a physical tool: a shovel, a hammer, an egg beater, a typewriter. These may be made of wood or metal or plastic, but each is a tangible object that you can see and touch. In addition, each is used to manipulate physical objects (e.g., wood, food, or paper). Problem-solving tools can also be *abstract* or nonphysical.

Abstract Tools.

The dictionary is a tool. But the tool is not the physical book; it is the information in the book. The catalog system at a library is a device for locating books. Until recently, the system has required a physical card catalog. But the system is really a methodology for categorizing texts into subject areas. The *Dewey Decimal System* (now largely replaced by the *Library of Congress* system) is a tool for this purpose. The system has remained the same, even though card catalogs have been replaced by computerized systems. The system represents an abstract tool; there is no physical item that you can touch and say, “Here is the tool.” Similarly, the categorization of tools is an abstract tool.

This course focuses on problem-solving tools. The most important problem-solving tools are abstract toolsconceptual tools that help you think your way to a problem solution. The approaches to problem solving or to using a computer are abstract tools. The procedures of arithmetic are tools for finding answers to arithmetic questions. When you learned to add or divide, you acquired a tool for solving a large, but specific, set of problems.

Abstract Objects.

Tools also manipulate abstract objects. Groups of characters, numbers, and words are all abstractions of real-world objects. In general, information is abstract. Computers are excellent tools for dealing with such abstract objects. While individual operations will always manipulate relatively concrete objects such as characters or pages, they will also manipulate the abstract information.

Abstracting About Instructions.

Give a little extra thought to the process of giving instructions. The user controls a hand-held tool directly (e.g., guiding a hammer to its target). The user controls a power tool indirectly by pushing buttons

or turning dials (e.g., guiding a car with a steering wheel). Sometimes these dials are labeled clearly and sometimes they require experience to learn how they work (e.g., steering wheels and brake pedals have no labels). Every object requires a command or instruction mechanism. You can often use an analogy to a concrete hand-tool operation to help predict the mechanism for manipulating the corresponding object in a computer system.

Abstracting About Lessons.

Notice that this text has introduced the concept or *model* of a computer almost entirely through analogies to other tools such as knives. Already you are using the tools of the discipline. An analogy is an abstract toolone that appears extensively in this text. An analogy allows you to think about one concept in terms of another. By thinking of the analogy, you can get a “rough idea” of how something works. Analogy is perhaps the most powerful learning tool that humans possess. As additional forms of software are discussed, use analogies to word processors to help build a picture of how they work. Much of this chapter builds a model of computing and writing. Be sure to notice how future chapters test that model and push it to its limits.

When can a Machine Help? When Not?

We live in a world of tools. Some are certainly more useful than others. I personally find many kitchen gadgets to be relatively useless. I do not need a tool to hold my bagels for slicing, a salad spinner, or an electric yogurt maker. Even generally useful tools are not useful for every possible circumstance. If you are about to beat two egg whites, it may be more work to get the electric beater out of the cabinet and the two beater-blades from the drawer and plug in the beater (and of course clean up when you are done), than to just beat the egg whites with a wire whisk. Similarly, it will be necessary to understand enough about computers to evaluate the effectiveness of the tool for a given task.³ And it should be no big

surprise at this point that much of this expertise will be gained through practice.

2.4

Participants in the Use of Tools

The discussion thus far describes three participants in any action: the operator or tool, the human user, and the manipulated object. There are also indirect participants in these actions for example, the person for whom an object is created, or the society in which it will be used. The preface describes computers as the machine tools of the information age. Because computers are tools that can create tools, the impact of an action using a computer may be considerably amplified. Two significant research areas did not become part of computer science until much later than other areas:

3. Actually, this text deliberately errs on the side of overuse but that should only compensate for the fact that many of you have probably erred on the side of underuse for most of your lives.

Human factors

Social implications

Whenever you use a modern computer you see the results of human factors research: computers are much easier to use now than they were just a few years ago, and they are getting easier to use all the time. For example, all modern systems use analogies to everyday (noncomputerized) activities to help users predict or remember how to use a program. While the design of user applications is largely the domain of professional computer scientists and programmers,⁴ social implications are everyone's concern. What impact will a new application have on society? Do computers change our concept of ethics? and What are the rights and obligations of the users and designers of equipment?

2.4.1

Ethics is a Problem?

This is a problem-solving course. Is ethics a problem? Apparently! As new users become comfortable with the concept of computing, they often focus on what is different rather than what is the same about this new world. One of the strangest manifestations of this new world vision is that people often assume that the ethical rules of the old world do not apply here. New users all too often seem unable to address even simple ethical dilemmas in the computerized world. Many chapters, in this book include short discussions of ethical problems that seem to confuse new computer users.

2.4.2

The First Rule of Computer Ethics

This text describes most aspects of computing as analogies of the traditional world. Ethics is no exception. In fact, the vast bulk of ethical dilemmas involving computers can be addressed through a single rule:

*If you would not do it without a computer
then do not do it with a computer!*⁵

Most computer ethics questions really reduce to a problem that you have already solved (a common technique in computer science: see Box 2.4). If you would not steal, do not steal using a computer. If you would not cheat on an exam, do not

4. However, starting in *Chapter 13: Programming in Applications*, you will find that your activities will begin to take on some of the characteristics of the designer rather than just those of the user.
5. Science fiction fans may want to compare this rule to Isaac Asimov's first rule of robotics:

A robot may not injure a human being.

(See *i, robot* by Isaac Asimov, © 1950 by Doubleday, Inc.)

Box 2.4 I've Already Solved That!

Reducing a problem to one you have already solved is a favorite technique of mathematicians. If two problems really are identical, why solve it again, if you can take care of the issue by showing the equivalence? This is a bit more formal than an analogy, but essentially the same concept. Mathematicians use the technique so often that it often becomes the punch line of jokes:

Q: How do you tell the difference between a mathematician and a physicist?

A: Ask them each two questions: “How do you boil water on the front burner of your stove?” and “How do you boil water on the back burner?” They both answer the first question with the same answer: “Put water in a pan. Put the pan on the front burner. Turn the heat on. Wait until the water boils.” For the second question, the physicist answers in almost the same way: “Put water in a pan. Put the pan on the back burner. Turn the heat on. Wait until the water boils.” The mathematician instead says: “Move the pan to the front burner. That reduces the problem to one we have already solved.”

cheat using a computer. It really is that simple. It seems surprising then that so many users find this confusing when they use computers.

Corollary.

The corollary to the first ethical rule of computing is even simpler than the rule:

Think!

Obviously you may need to stop and think about the problem: How is this new scenario like the more traditional world? What would I have done there? What does that imply I should do here? Each is a

very simple problem.⁶

As with any other tool, computers make it easier to perform a task they do not alter the ethics of the task. Guns make it easier to kill people; they do not make it more or less ethical. Slogans such as “Guns don’t kill people. People kill people!” do not represent debates about the ethics of murder, just the mechanisms

6. Once again: divide the problem into subproblems and conquer it.

for preventing murder. The nature of the crime does not change if it is committed with a gun or a computer, only the ease with which it can be committed. Think!

Future chapters will address specific issues as they arise, but almost all issues can really be addressed by the first ethics rule of computing.

Exercises

2.8 Create a list of academic tasks for which you believe computers are not likely to be useful tools. For each, give the reason you think computers would not be useful. Do not turn this assignment in, but keep it for your own future reference.

2.5

Summary

Three basic tools for problem solving are:

Help Save Undo

Trying it may be the most important step in learning to use a new tool. Abstract tools enable computer scientists to understand problems and their solutions. Four of the most important abstract tools are:

abstraction questioning

analogy experiment
 (trying)

Important Ideas

revert scientific method backup

onlinemain (internal) revert
memory

meta- abstraction analogy
scientific method

3

Documents as Objects

The whole is simpler than the sum of its parts.

Willard Gibbs

Pointers

Gateway Basics

Lab:

Lab Unit 20: Crisis

Manual: Management

3.0

Overview

Most of the discussion of tools thus far has focused on tools for creating or modifying an object. Other tools do not so much modify an object as change its situation. For example, if you cut and paste a text segment, you have changed the document. But if you move the document from one location to another, you have only changed its situation; the document itself remains the same. This chapter focuses on tools that perform actions with complete objects (such as moving an object), rather than on parts of objects. Essentially, every electronic tool provides basic high-granularity capabilities that change the situation or state of entire documents, not just parts of documents.

3.1

The System and High-Granularity Operations

Most of the actions described in Chapter 1 were low granularity: operating on one character or at most a few characters at a time. The actions manipulated a small object (the character) as part of a larger

object (the entire document). Each application did support a few high-granularity actions such as Save and Print. There are actually many uses for high-granularity actions, but most of them are usually executed from outside of the applications at the *system level*.

3.1.1

Operations on Documents

Every system provides document-manipulation tools analogous to those used for paper documents. Most of these actions seem familiar: they are similar to low-granularity actions performed on text.

Moving.

Any document can be *moved* to a new visual or logical location. The new location can be in another folder on the same disk or on a new disk. By *moving* individual disks or folders, the user can maintain the organization of the entire desktop.

Copying.

Any logical object can be copied. The new *copy* can be placed on the same or another disk. Copying is useful for making backups, or for creating two similar versions of an object, for example:

A history of your activities. If an instructor says, “I never received your assignment #5,” your electronic version of that document is powerful evidence in your favor.

A template for a future similar document. A letter to your mother could form the basis of a letter to your grandmother. If you plan to tell them both about the same event, you might be able to reuse much of your typing.

Deleting.

A user can throw away any document. This frees up space on a disk and reduces clutter. *Delete* an object when you are sure you no longer need it. Remember, though, that keeping an object is relatively inexpensive. Always consider possible future uses of the document.

Opening.

Looking at a paper document requires opening itremoving from an envelope, unfolding, and so on. Computer documents can also be opened. That is, you can look at their content. Notice that we usually say *Open* an existing document, but *create* a *New* one.

Renaming.

The user can change the name of any document by *renaming* it. Select names that help you understand what is in the document.

Searching.

Just as a word processor allows the user to find a word or other sequence of characters, the system provides tools to help the user find an object. For example, a user who has recently reorganized documents may not remember the exact location of every single one. Searching tools provide a significant advantage over traditional human search techniques. Computers are very good at tasks such as finding documents. Compare the task of looking for that homework assignment with asking a computer to find it for you. The most common name for searching is *Find*.

Visualizing.

The user of a document views that document as a single entity or a complex entity depending on the correct action. Similarly, users may want to visualize or organize objects based on various properties: name, size (find the large ones to see what used up all the space on the disk), type (find all the documents created by a given word processor), age (you know you created it yesterday, but have no idea what you called it).

3.1.2

System Commands

The highest level human-computer interface is often called the *system* or the *operating system*. Actually, the system does much more than provide the human interface, but that is all that is visible to a new user. Applications are to the system as individual data items are to the application: they are operands. The user can manipulate an application by giving commands to the system. In fact, you have already done this. The Open command is really a request for the system to open a document or application for the user. Generally, system commands operate on an entire document or application, while an application manipulates a portion of a document. The system acts as a high-granularity application: the objects it manipulates best are documents and other applications. An object is a part or a whole, depending on the user's perspective.

3.1.3

Finding the Commands

Generally, the high-granularity operations such as duplicating or finding files are either grouped together in a single menu (usually File) or performed by direct graphical manipulation (e.g., dragging an icon). See Figure 3.1.

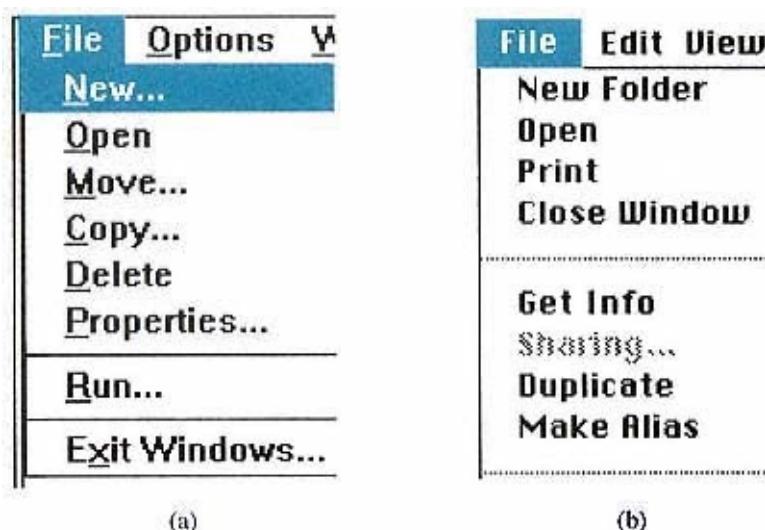


Figure 3.1
File Menu for Large-Granularity System Commands
(a) Windows,
(b) Macintosh

Exercises

- 3.1 For each of the above file-manipulation instructions, describe the closest text-manipulation operation.
- 3.2 List as many “duplicate” instructions as you can. That is, find and list actions that can be performed from within an application you have already used and also at the system level.
- 3.3 Although the menus in Figure 3.1 are very similar, there are differences. For each command not present on your system, form a hypothesis about what that command might do and figure out how to accomplish a similar result.

3.2

Understanding Memory

The use of many of the system commands, like `Print`, is clear. But a general understanding of the manipulation of entire applications requires some understanding of *memory*. Humans have at least two forms of memory: *short term* and *long term*. Roughly, the former is used to keep track of what you are doing right now (a classic short-term memory experiment asks the subject to repeat back a list of 5 to 10 numbers). Long-term memory holds information that can be retrieved at a later date. Certainly your name and address as well as your class schedule fit into this category. In general, short-term memory is thought to be quite small (see Box 3.1), and volatile, while long-term memory is much larger. In a very real sense, computers also have short- and long-term memory.

A computer’s short-term memory is called *random access memory (RAM)*. It also may be called *main memory* or *internal memory*. A computer can fetch information from RAM very quickly; typical desktop computers can fetch a million or more characters per

second. But it has a disadvantage: because RAM is electronic, it is very volatile. Information remains in main memory only as long as the current application requires it. From the user's perspective, this means while it is actually in use, or while the document is open. Thus, when a user creates a new document, that document lasts until:

- the user closes the document,
- the user closes the application,
- the user shuts down the computer,
- the power is interrupted, or
- a major (called *catastrophic*) error occurs.

That is, as long as the current document is active. The first three items in the above list are under the user's control, but the last two are not. And that is the problem: when an interruption such as a power failure occurs, the content of main memory is lost.

A person who wants to remember a fact usually writes it down on a piece of paper. Most people know they can't keep every needed fact in their heads. Neither

Box 3.1 The Magic Number 7!

G.A. Miller showed in a famous 1956 paper that human short-term memory could only hold approximately 7 items. Short-term memory refers to those things a person can remember without memorizing them. So when given a list of unrelated numbers, most people can correctly remember only seven items in the list. Those who can remember more do so by “chunking” the numbers into larger groups. For example, one person might remember the entire list 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 9 by grouping the digits 1, 4, 9, and 2 into a single easily remembered number (there are at least two more such combinations in this series). In contrast, with rehearsing, people can remember an arbitrarily large number of items. This work has had significant influence not only in the field of psychology but also in the computer science subdiscipline of artificial intelligence. (See “The magical number seven plus or minus two: some limits on our capacity for processing information” in *Psychological Review* 63, 81-96.)

can a computer. Eventually, voluntarily or not, a given document will close. A user who wants to keep the information (the usual case) must copy the information to *long-term* or *external memory* (a slightly misleading term: long-term memory is called external, even if it seems to be housed in the same cabinet as the computer itself). Placing a document into long-term memory is actually what the Save command does. (See Section 2.1.3, “Save.”) The most common long-term memory is a magnetic medium, usually a *disk*, but sometimes *tape*.¹ Magnetic media work on the same principle as audio tapes: they contain a magnetic representation of the information. A read-write head can record (*write*) the information and *read* it back later.² The most important feature of magnetic information is that it is (relatively) nonvolatile compared to the electronic RAM. When a computer is turned off, the magnetic information continues to exist and can be read at any time in the

future by a computer (the same or another). The

1. Magnetic tape is becoming much less common. Its principal use today is for large archival backup systems. For example, a computer center may back up all of their disks onto tape.
2. Notice that the verbs *read* and *write* refer to the perspective of the computer, not the disk. The computer reads information *from* the disk, and writes information *to* the disk not the other way around.

disadvantage of long-term memory is that it is very slow compared to RAM. Accessing a single character from a disk could take as much as 1000 times longer than accessing the same character from the short-term memory.

3.2.1

Classifying Long-Term Memory

Terminology for computer long-term memory can be confusing. All long-term memory is logically equivalent; the popular names often refer to physical attributes of the device.

Disk and Disk Drive.

Information is held on the disk itself. The *disk drive* both turns the disk physically and reads and writes information to the disk. The distinction is analogous to record and phonograph or to tape and tape deck. This distinction can seem especially vague when disk and disk drive appear as a single physical unit, often housed entirely within the computer cabinet. The terms, *internal* and *external*, indicate whether or not the entire disk drive is contained in the same cabinet as the computer.

Removable Media.

Some disks (and all tape) can be removed from a computer. The most common removable disks are called *floppy disks*. The principal advantage of floppy disks is that a user can remove the disk and take it away. This has several important implications. The user can take the data to another computer; if another user uses the same computer (the typical situation in a college lab setting), that new user cannot see the private information and data of the first user. Removable media are inexpensive (less than \$1), so incremental costs are small. Unfortunately, they hold relatively little information (typical floppy disks hold about one million characters or *bytes* also called one *megabyte*, see the naming conventions in Box 3.2). The exact capacity varies, depending on the specific

model of *disk drive* or computer. This sounds like a lot of characters about as many as a novel. Data other than simple text (e.g., pictures) can use up the storage much faster. However, one floppy disk will often hold all the information a student generates in a one-term course (but remember that backups will require additional disks). The icon for a disk looks like a floppy disk (e.g., see the *floppy disk* icon in Figure 3.2 on page 46).

Nonremovable Media.

Most computers have another form of long-term memory: the *nonremovable* or *hard disk*. Unlike floppy disks, a hard disk is a

3. RAM and external memory are just two points on the traditional computer science classification of memory, called the *memory hierarchy*. The usual order of the hierarchy lists the memory forms in order of both decreasing speed and decreasing price. All but RAM and external memory are invisible to most end users. *Hierarchy* is a term that will appear again in this and later chapters.

Box 3.2 Anachronisms Revisited

Floppy disks The name *floppy disk* is now largely a historical accident. The earliest disks were rigid, and permanently mounted in a cabinet. A smaller, cheaper disk was developed for use with personal computers and promptly nicknamed “floppy” because the flexible disk drooped if held by one edge. Newer, even smaller removable disks in a rigid case appeared in the mid 1980s, largely replacing the original floppy disk. Because they are used for the same purposes as floppy disks (removable from the machine by the user) these disks are often called “floppy” in spite of their rigid case. Indeed, if you were to break the case open, you would find that, inside, the disk is still floppy.

Big names Computer scientists generally use the Greek prefixes, *kilo* (Greek: *khilioi*: 1000) and *mega* (Greek: *megas*: large), to mean the specific numeric values 1,000 and 1,000,000 respectively (but see Exercise 16.12). *Mega* is never used in its colloquial sense: big. Similarly, *Giga*: comes from the Greek, *Gigas*, giant. And yes, machine designers are planning ahead: one thousand gigabytes (one trillion bytes) is called a *terabyte* (from the Greek *teras*, monster). Probably no desktop computer yet has a disk of that size.

self-contained unit. The disk itself is permanently mounted in a box or case, along with its *drive* or motor. The case may be the same case that houses the computer itself, or a separate unit. Hard disks have two major advantages. First, they are significantly larger than floppies; typical capacities range from 20 to 200 megabytes (Mb); some are over 1000 Mb (one *gigabyte*, or 1 billion bytes). Second, hard disks are faster than floppies (but still considerably slower than main memory). The price for this speed and capacity is added cost and reduced transportability.

Removable and nonremovable storage media are logically

equivalent; both are long-term external storage devices, distinct from internal memory. From the user's perspective, storing information on one is exactly the same action as storing it on the other: all operations are identical.

3.3

Understanding Files

The physical location of data is not as important as the *logical* organization where the data appears to be stored. The user need not know where on the disk a

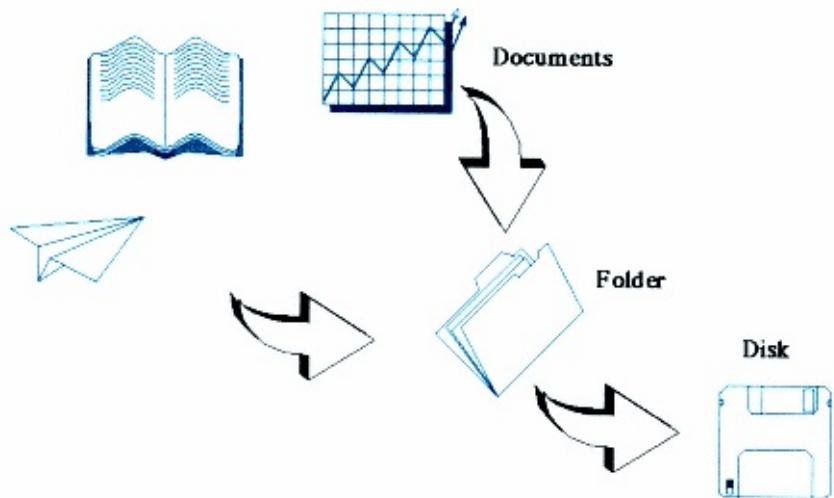


Figure 3.2
Organizing Files: The Desktop

file is physically located; the computer can keep track of that. But the logical locations such as in a particular folder is essential. The user need only think in terms of logical locations. Normally the user keeps logically related items together (e.g., in the same folder). Understanding these logical relationships between files and their containers is essential.

3.3.1

Types of Objects

Files.

Documents and applications are collectively called *files*, as in “the file cabinet.” Each document icon visible on the screen corresponds to a file. The user can manipulate the file (either document or application) by manipulating its icon.

Folder or Directories.

A *folder* or a *directory*⁴ is a place (like a manila folder) where you can place items such as files. Electronic *folders* are containers that can

4. *Directory* is the older term, which reflects the underlying

representation (invisible to the user). The directory actually does not contain any files at all. It only contains pointers (see the first page of this text) to the files. That is, it is a directory in the same sense that a phone book is a directory: it directs the computer system to the files. This text uses the newer term, *folder*, which better reflects the user's perspective.

hold electronic objects: documents, applications, and other folders. The screen icon for a folder looks like a file folder.

Disks.

A *disk* or *hard disk* is a physical device. Generally it also serves as a logical container: each file or folder must be entirely contained on a single disk.

Desktop.

The global object that holds everything else is called the *desktop*. Like a physical desktop, it contains all of your documents and tools. Visually, the desktop is the entire computer screen. While both disks and folders can hold folders, only the desktop can hold a disk. Figure 3.3 schematically represents the possible organizations of files and folders.

3.3.2

Organization

Each user can create an individualized organization scheme. For example, one user might want to save an application and all of the documents created by that

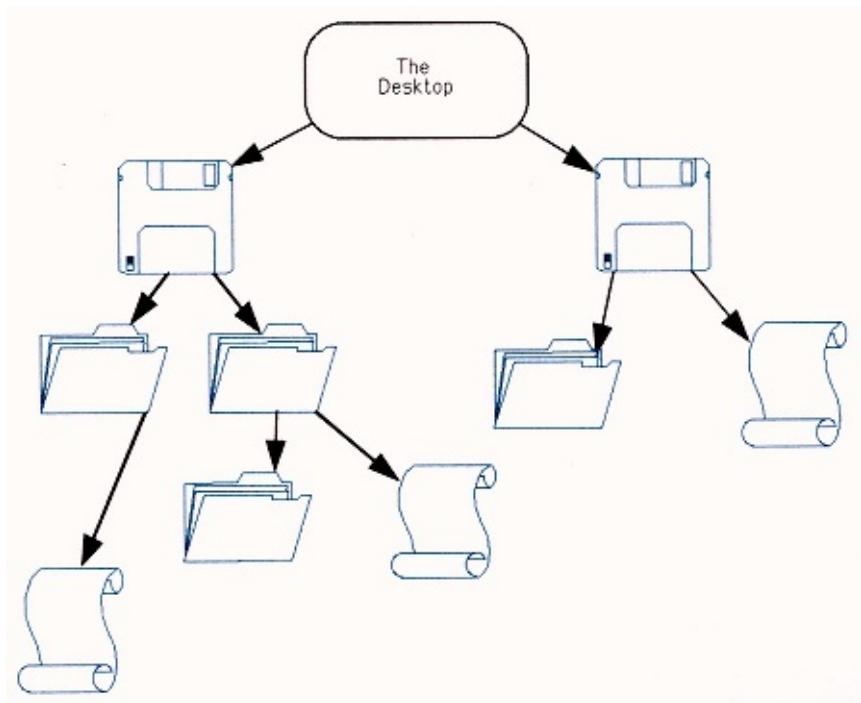


Figure 3.3
Folders and Files

application in one folder. Another user may prefer to save all homework in one folder, all experiments in another, and so on.

Folders in Folders.

Think about the significance of the fact that a folder can hold a folder. What happens if the inner folder also contains a folder? And if that one does too? The *hierarchical* nature of organization is a theme that will recur throughout this text (for example, see Section 6.2, Hierarchical Organizations).

3.3.3

Naming Files and Other Objects

Low-level objects do not have individual names (the fifth character in the third row?). But higher-granularity objects such as files and disks do have names, usually given to them by the user. Even though these names are the only handles by which you can access these objects later, novice users often use very poor naming conventions. The primary problem with many names is that they are too short to adequately describe the object. There are two principal root causes. First, until relatively recently, many computer systems did not allow long names. Fortunately, most systems now allow names as long as any user is likely to type. Similarly, some systems did not allow special characters such as blanks as part of file names. Many users simply developed their bad habits when there was no alternative. Since the old restrictions on names are now gone, make it a point to create good names.

In addition to the system-imposed restrictions, people seem to be inherently lazy.⁵ At first glance, typing a short name seems to take less time than typing a long name does. Similarly, the creation of a useful name may require the user to stop and think for a moment before creating the name. But poor naming conventions ultimately waste considerable time. Typing a long name may require a few seconds more, but opening the file to see what is in it just once will

use more of your time than you lost originally. More importantly, losing track of a single file will also use more time than you lost in the typing. Always name files and other objects very carefully.

Some Naming Conventions.

Selecting good names comes with practice. The following list of conventions will help you get started:

The name should describe the content of the file.

The name should distinguish this file from related or similar files (e.g., `homework` may describe several different files you create this semester).

There is little reason to include your own name as part of the file name unless you will be giving the file to someone else (e.g., turning it in to an instructor, in which case it is essential).

5. This characteristic certainly applies to most computer scientists. But, by recognizing it, we use it to advantage. Because we see that a bad file name will eventually cause us more work, we try to name it well the first time.

Names like ABC or 123 are of little use unless they help distinguish between versions.

Names of friends (e.g., boy or girlfriend) or hobbies are not useful unless the file relates to that person or object.

A name should be easy to read. Do not over abbreviate or clutter with strange punctuation.

Use spaces and punctuation to help make names more readable.

A name composed of both upper and lower case letters (as in everyday English) is easier to read than one in all upper case letters.

Use well-named folders to help maintain the file's content.

3.3.4

The Desktop as an Object

Consider an alternative view: rather than viewing the file as the central object that you can modify with low-granularity actions or change the state of with highgranularity actions, view the desktop as the central object. Low-granularity actions on the desktop change individual parts of the desktopthe files. Deleting a file is to the desktop as deleting a character is to a file.

If deleting, creating, copying, and moving files are low-granularity actions with respect to the desktop or the system, what would the high-granularity actions be? Anything that impacts the entire desktop or system, such as:

changing the way you (the user) view the system

quitting the system (or the machine)

reorganizing (cleaning up) the desktop

and so on.

Exercises

- 3.4 Describe the difference between opening a document and opening an application.
- 3.5 Suppose you were organizing all of your class documents for every class you have taken. How would you organize them? Suppose you had indexes for the files based on other organizations, and you could not remember the title of a paper you wrote. What indexes would be useful for finding that document under what circumstances?
- 3.6 Try creating a hierarchical set of folders: create one folder and put two new folders inside of the first. Put two folders in each of those and so on.
- 3.7 Using the tools described in this chapter, can you find a way to complete Exercise 3.6 much more quickly and easily?

3.3.5

Applications as Data

A computer application is itself information: a description of how to manipulate documents. This is in fact one of the defining properties of computers they are

sometimes referred to as *stored-program computers*. For the moment, do not worry about just how an application is stored but do observe that it is stored as data. This means that a computer can read, maintain, or write a program in exactly the same way that it manipulates user documents. As a result applications can be moved, copied, deleted, opened, or closed. Opening a document makes the content of the document visible; opening an application makes the tools of that application visible.

Exercises

- 3.8 The desktop analogy is really very apt for many operations. Think about the following actions/activities that you may wish to perform at a desk and find the closest analogy in your computing system:

throw an object away

move an object from one place to another on the desk

put an item in a file folder

look a word up on the dictionary

look at a particular object (of many on the desk).

Remember: as with noncomputerized actions, every computerized action requires both a document and an application to manipulate the document.

- 3.9 Many applications contain (high-granularity) operations that duplicate the function of system operations (which must be low granularity). Find at least three such operations in an application you have used.

- 3.10 When would you want to perform an operation from within the application (as high granularity), and when would you want to perform it from the system level (as low granularity)?

3.4

Tools and their Creators

Most of this chapter and the previous chapter discuss tools from the viewpoint of the tool user. The tool *designer* or inventor also has an important perspective, but many new computer users are initially confused about the roles and the rights of the creators of the tools they use. Most users have little problem understanding the rights of the manufacturer of the computer they use in their lab: the user (or the institution) must pay for that computer. But software can be more confusing.

3.4.1

Ownership:

Who Actually Owns Software?

Normally a software author sells a *license* to use a product not the product itself. Perhaps a better analogy is renting. When you rent a floor sander, you really buy the right to use the floor sander for some period of time. When you buy software you actually buy the right to use the software within a set of restrictions. Usually

the restrictions explicitly state that you do not have the right to copy the software for the purpose of giving or selling the copy to another person. Obviously, software licenses almost universally include the right to copy for one important purpose: making a backup.

3.4.2

Intellectual Property

In our culture, we all understand that the manufacturer is entitled to compensation for the effort needed to manufacture a computer or any other product. But many people seem confused about *intellectual property* such as the applications they use on that computer.

Example 3.1 Mary purchases the *PrettyPrint* word-processing program. She pays a fair price and her purchase is in all ways completely legal. She brings it back to her dorm room where she shows it to her roommate Juanita. Juanita is very impressed and asks Mary to let her copy the program. May Mary let Juanita have a copy?

For many people the answer is yes. After all, they reason, Mary has paid for the program. It is hers to do with as she pleases. Unfortunately, this is not correct, either legally or ethically.

Intangible Assets.

Intellectual property is intangible. The price of a book represents not so much the paper for the pages as it does the costs of creating the ideas written on those pages. While few people would steal a computer, many are willing to steal software or music by copying the recording. It is not the cost of the medium (tape) these people are stealing, it is the cost of the intellectual property. Most adults understand this concept within limits; they would not sneak into a motion picture theater, concert, or sports event without paying the admission fee. They recognize that the performers are entitled to

compensation for their work. Exactly the same principle applies to computer software: the creators are entitled to fair compensation for their work.

Oddly enough, it may be the computer itself that confuses some people with respect to copyright responsibilities. A radio, or computer, is not easily created. And there is no easy way to "copy" a radio. Yet audio tape recorders and computers both make it very easy to copy intellectual property (music or software, respectively). The ease of copying seems to confuse many users. "Can anything so easy to do be immoral?" they reason. But it is not the ease, but the principle of just compensation that determines the answer here.

Patents and Copyrights.

Since a manufacturer who develops a new product or even a part of a product is entitled to reap the benefits of that idea, it may apply for and receive a *patent* on the product. In granting the patent, the issuer (e.g., in the U.S., the Patent Office) certifies that the patent does indeed represent an original and unique idea, concept, or product. Any person wishing to

manufacture a product incorporating the patented feature must pay a royalty to the inventor. This assures that the inventor will receive some reward for the effort expended to develop the product.

A similar concept applies to computer applications or software. The creator of computer applications may receive a *copyright* the same sort of protection enjoyed by authors, musicians, and film makers. Like a patent, a copyright applies to *intellectual property*. The copyright represents an assurance that the creator of the product is entitled to some sort of benefit from the use of that product. As with physical products, the price of copyrighted material includes compensation for the development costs of the product.

Amortization.

Some small portion of the price of any product represents the cost of materials. Other portions represent the cost of labor for the actual manufacture of the computer, the cost of research and development needed before the machine could be manufactured, and the cost of advertising. Finally, some is profit to the manufacturer. Most of these costs are not directly attributable to the specific computer that sits on your desk. Instead the total costs are *amortized*⁶ over all machines sold (or expected to be sold). That is, if the company spends \$100 million for research and development and expects to sell one million machines, the price of each machine must include \$100, representing the appropriate portion of the development costs for the machine. No author could afford to write books for a single reader. For example, an author who wrote one book a year would have to sell that book for \$25,000 in order to live on the proceeds. Instead she sells copies of the book for \$10. If she can sell 2500 copies she makes the same income a much more likely scenario. The author depends on all of the readers of her book helping pay her for her effort. Similarly, a major computer program may require hundreds of thousands of hours to develop. Few users could afford such software if it were not for amortized costs. Like the price of a

music album, a theater ticket, and a book, software must include such amortized costs.

A person who steals a book, steals from the author. This is true whether that copy is a regular edition stolen off the shelf in the bookstore, or a copy made on a home copying machine. Copying a book is theft, just as taking a sculpture or mousetrap would be theft. Likewise, copying software is theft. Protection for designers assures that new (and better) software will be created. Without that protection, few people would devote the hours and dollars needed to create quality work.

3.4.3

Common Misconceptions

Many people justify their violations of the above arguments on the basis of reasonable sounding arguments, such as the two discussed here.

6. Laboratory Unit 10.4 contains a contrasting example of amortization.

“The Owner Didn’t Lose Anything.”

Some new users get confused because the copies they steal don’t exist until they steal them. They are not taking anything the author already has or needs. Would the same apply to a car the owner was not using at the moment (especially if you intended to return it)? Remember that you are taking something from the authorher intellectual property.

“I Wouldn’t Use it if I Had to Pay for it.”

The police would give you little sympathy if you explained that you stole a car because you couldn’t afford it; the owner really didn’t lose anything, since you would not have actually purchased it if you didn’t steal it.

The bottom line for most such arguments can be found in the first principle of computer ethics: “If you would not do it without a computer then do not do it with a computer!” (See Section 2.4.2, The First Rule of Computer Ethics.)

Exercises

3.11 Suppose you are employed at Acme Tools. You have a copy of the *PrettyPrint* word processor at the office. Your home computer is compatible with the office computer. Can you copy the software for your own use on your home computer? Defend your answer.

3.12 Suppose you bought a new office computer. Could you copy the software in the above problem to the new computer? Always? Never? Only under some circumstances? Defend your answer.

3.5

Summary

Files (documents and applications) can be manipulated by the user in ways analogous to the manipulations of individual characters within a document. The system provides a uniform and consistent user interface for these capabilities.

Important Ideas

open close print
save move duplicate
system copyright intellectual
level property
amortize application desktop

Credits.

Many of the figures in this text were clipped from collections of predrawn pictures, called *clip art*. Individual items in Figures 3.2 and 3.3 are from the *HyperCard* stack version of the application *ClickArt®*, and from *Framemaker*.

ClickArt® is a registered trademark of T/Maker Co.

Framemaker is a registered trademark of Frame Technology Corporation.

4

Sharing and Communicating Information

Nothing is more fairly distributed than common sense: no one thinks he needs more of it than he already has.

RENÉ DESCARTES¹

Pointers

Gateway Mail systems

Lab:

Lab Unit 3: Electronic

Manual: Mail

4.0

Overview

When you create a document by hand, you don't create it just to create it. You want to do something with it. In very broad terms you have two choices. You can:

give it to someone else, or

keep it for yourself.

The same is true for documents created with a word processor or other computer application. Many computer tools (e.g., the word processor) seem designed either to keep a document or create a paper copy to be delivered the old-fashioned way. You can give the document to someone by putting it in the mail, taking it there yourself, hiring a messenger, or any of several other methods. You may keep certain private documents to yourself, such as diaries, a collection of your personal poems, notes for an upcoming term paper, or records for calculating taxes. Some documents may

require both actions saving and giving away (e.g., paying a bill,

1. French philosopher, mathematician, and scientist (1596-1650), the founder of analytic geometry and designer of Cartesian coordinates. The quote is paraphrased from his *Discourse on Method* (1639).

but keeping a personal record). Clearly, keeping an item requires a place to keep ita file cabinet, notebook, or shoe box.

4.1

Shared Data

Something there is that doesn't love a wall, And wants it down.

ROBERT FROST, *Mending Wall*

With the exception of Print, most of the tools discussed so far seem to be better for keeping data for yourself than for giving it to someone else. But most problems require some means of communicating the solution to others. Printing creates a paper copy that you can deliver to another person. Similarly, you can make an electronic copy of a document on a disk and hand the disk to an associate. Both of these solutions seem insufficient in an electronic environment. Surely, if one person has some electronically stored information and another person needs access to that information, there must be an electronic way of sharing it. Most computing systems provide built-in tools for this purpose. Two of the common tools are:

file servers for sharing information among groups of people

mail systems for delivering information to specific individuals.

Each tool has its own characteristics, its best uses, its advantages, and its disadvantages.

4.2

File Servers

In addition to removable and nonremovable storage, there is one more common form of storage: the *file server* or shared disk. A file server is essentially a disk accessible to many users from many machines (called *clients*). A musical recording owned by one individual and kept at the individual's house is really only available to that individual. A recording owned by a radio station can be

broadcast to any number of listeners. Similarly, a file server can be accessed by many users using several machines. Once a user has accessed a file server, it appears exactly the same as any other disk. But unless the users also use the same computer, they need a means to access that disk.

4.2.1

Computer Networks

Use of a file server does impose a few additional requirements. The most obvious is that the user's computer and the file server must be somehow connected. Typically, both machines are on a *local area network (LAN)*. This means that they are connected together by a wire or cable. This same LAN probably also connects an individual computer to a printer. Figure 4.1 shows a LAN which includes at

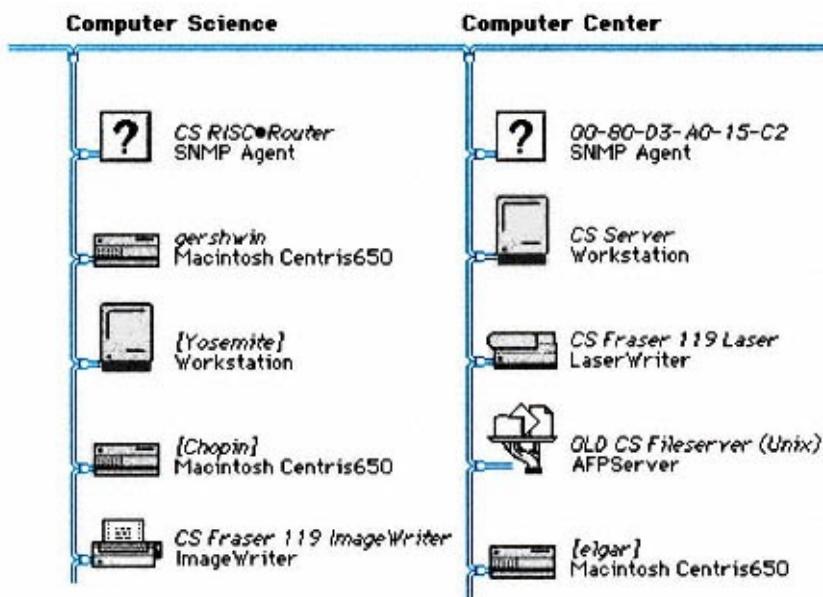


Figure 4.1
A Small Local Area Network

Source: The images in this figure were originally created using the Shareware program *Trawl* by Michael Love.

least two kinds of computers, two kinds of printers, and a file server.² A networked computer can take advantage of resources stored on other machines on the network. The word *local* distinguishes the network from *wide-area networks*, described in Chapter 22: *Telecomputing*. Usually, local area nets are restricted to a building or campus.

Access Privilege.

A less obvious restriction is *access privilege*. Consider what would happen if ten people all tried to write onto the same file at the same time. Would the result contain all of their work intermixed; or would each successive user completely replace the work of the previous one? Neither possibility seems to be a very good situation. Therefore, most file servers restrict the use of individual users. One common restriction is *read-only* access. Each client computer can

2. This particular network is divided into two logical divisions or *zones*, with routers to move messages between the zones. Dividing networks

into zones is a common technique for both reducing interference between messages and restricting what individual users can use.

read but not write data on the server's disk. There is no problem if two computers attempt to read at the same time.

Semiprivate information also poses some questions for shared disk users. For example, a corporation may wish to restrict users other than its own employees. An academic department may want to make tests and grades available to faculty but not to students. Most file servers can restrict access to such files, usually through a *password* protection system. In that case, users must provide passwords to prove that they are authorized to access the data. Often users have differing levels of access privileges. For example, many people will be allowed to read data, but only a single user will be allowed to change that data.

Finally, although a personal computer can recognize any disk connected directly to it, file servers offer a very different problem. What if a network has 10 publicly readable file servers? Should each computer find each one? That is, should all 10 servers be as accessible as local disks, or should the user request specific devices? Generally, accessing a server requires the user to *mount* or connect to the server that is, establish a connection between the user's machine (client) and the server. Generally this is done once for each server the user wishes access to, and any required password verification is performed at the same time. You may think of connecting to a file server as opening the file server.

Exercises

4.1 The method for accessing file servers varies more than do most of the computer tools discussed so far. Find out if there is a file server available to you and determine how to access that server.

4.2 Copy a document from the server to a local disk. Determine if you are allowed to write to the server: try to copy a file from your local disk to the server. (Be polite: remove any test data

copied to the server when you are done.)

4.2.2

Danger:

Viruses

One down side of shared data is the *virus*. Like their biological namesakes, viruses are capable of replicating themselves in new files and even new systems. A person who never comes in contact with other living beings has little or no chance of catching a virus. Similarly, as long as a user only uses one machine (including its disks) and that machine (and its disks) is used only by one user, there is little or no chance that a virus can get into the machine. But as soon as data is shared either via a file server or by handing a disk to another user viruses become a real possibility and every user must be wary. Note that sharing a floppy disk is probably the most common method by which viruses are spread. Most owners of file servers have taken actions to protect their server against viruses.

The concept of a machine catching a virus or disease sounds like science fiction. Actually the concept is very simple (although the details can be a lot of

work). Although they have many similarities to their namesakes, computer viruses are not a product of natural selection or evolution. They are the work of particularly malicious human beings, who modify an existing computer application. The human creates the virus and pastes it into an application, say a word processor. The basic logic of the virus itself is:

- Perform some simple “signature” action.*
- Make a copy of yourself (the virus).*
- Paste that copy onto some other application.*

Thus, when an unsuspecting user runs an infected application, the application does everything it normally would and more. At some point (perhaps when the user opens a document), the virus performs its three steps. The signature is a sort of “Kilroy was here,” the virus creator’s way of showing off that she foiled your defenses. The signature may be fairly innocuous, such as a beep or a message of the form, “Ha Ha! I got ya!” Or it may be incredibly destructivelike deleting all of your files. The last two steps are really just a “copy and paste” similar to ones you have performed on text or entire documents, but they represent the real power of the virus. The virus may paste itself³ into any other file on any disk accessible to this computer; that document need not even be open. Thus, when the virus appears on a new document, it is almost invariably a surprise.

Protecting Yourself Against Viruses.

Anti-virus software is readily available. Several vendors and individuals feel so strongly about viruses that they have made their products available for free. As your first line of defense: get an anti-virus tool and use it regularly. Many run automatically; once they are installed you never need to take an action again unless you actually have a virus. Most multiperson facilities (such as computer center labs) have installed anti-virus software on their own equipment. In addition, take extra care whenever you obtain

software that may not be guaranteed virus-free. This certainly includes distant file servers, any “free” software or unauthorized copies, and disks that have passed through many hands.

The Perpetrators.

When any new form of dastardly deed appears, there is a period of time before the community decides what constitutes suitable punishment. Although some perpetrators think of their work as a harmless prank and others see themselves as self-appointed police, patrolling the computer labs for people who have not backed up their disks (and teaching them a lesson), the general consensus in the computing community is that the creators of viruses are essentially criminals. Several virus creators have now been sentenced to prison.

3. Notice that “copy itself” represents a recurrence of the self-reference concept mentioned in Footnote 1 of Chapter 2. This concept is truly ubiquitous.

Exercises

- 4.3 Suppose some individual spends her time looking for unlocked dorm rooms. When she finds one, she enters the room and breaks whatever she can put her hands on easily. Compare the acts of this individual with one who creates a virus.
- 4.4 Compare the self-reference of a virus with that of a Help command. List other examples of self-reference mentioned in this text.

4.3

Mail Systems

A file server makes data available to many people. In contrast, a mail system allows a user to send information to one specific user. Historically, an individual writes a letter, puts it in an envelope, and mails it at the post office. You could do the same thing with a word-processed document by printing it. That costs \$.32 plus a few days. In this information age, there is a better alternative for many purposes: *electronic mail* or *Email*, for short. Email has many advantages over its physical analog: it's faster, it's more flexible, and (if you happen to have a computer network handy) it's even cheaper. Mail delivery does require a delivery system accessible by both sender and receiver, but that is rapidly becoming less of a problem.

Sending a document is a high-granularity operation: the operand is an entire document. Thus, it is not surprising that mail systems often seem to have similar capabilities to those of the system itself. Email is like a composite of regular U.S. postal service or paper mail (sarcastically called *snail mail* by computer scientists), a phone answering system, a word processor, and the system. Consider the requirements on a traditional mail service. Some are easy to state:

Deliver mail from you to anyone else.

Deliver mail from anyone else to you.

Some are a bit trickier:

Return incorrectly addressed mail.

Help you find addresses.

Ship parcels as well as letters.

Forward mail if you have moved.

Take messages.

4.3.1

The Post-Office Model for Mail Tools

Like working with any new tool, understanding an electronic mail system may require that you view it from multiple perspectives. First, consider an Email system as a provider of basic post office services. Every electronic mail system includes the fundamental services analogous to those provided by the post office.

Send a Message.

This may be the most fundamental action. A mail system allows the user to send a message to another user.

Receive a Message.

Every Email system will receive messages. Unfortunately, some students get confused by this particular capability; they want to instruct their machine to receive a message just as they instruct it to send one. But you cannot tell the Email system to receive a message any more than you can tell someone to receive a letter from the U.S. Post Office. What you can do is tell the application to *check* for mail for you. Your system must check with the “post office” to see if you have mail. Most systems will do this automatically.⁴

Forward a Message.

If you have a letter in your hands, you can forward it to someone else. This requires exactly the same tools as sending the letter; the user must possess the letter, provide an address, and ask the system to deliver (forward) it.

Packages.

The U.S. Post Office delivers packages as well as letters. Most Email systems can also send more complicated documents by *including* or *attaching* them to a message. Clearly there is no way to send a box of chocolates, but you can include any file that you can manipulate at the system level. Sending a file requires special “packaging” so that the computing system can tell that it is delivering or receiving something other than simple text, but many mail systems will do this automatically.

4.3.2

Mail Tools Based on the File System Model

A user who receives many messages soon discovers a familiar problem: How can you maintain and organize large collections of individual documents? This problem is essentially the same as that of organizing a large collection of documents. Because a mail system is a self-contained document processing system, most Email systems contain analogs of the most common file handling tools (see Section 3.1.1). In many respects, the system makes an excellent model for understanding Email. An Email tool manipulates many messages or documents. A user may want to perform low-granularity actions on a document such as editing the text, or finding a specific word within that text. On the other hand, the user needs to manipulate that document as a single object to Save it, Print it, etc.

Folders or Directories.

Mail is a collection of electronic messages, similar to other documents. As such individual messages can be filed in an analogous

4. Some systems use a different principle: the post office delivers mail to the client whenever it can. This means the client program is receptive to receiving mail. The basic issue is unchanged: the user cannot command the client to receive mail.

manner. A user may file messages into individual folders, making it much easier to organize and keep track of these records. Records may be organized by grouping them logically into directories or folders, for example, all correspondence with the instructor in one folder, correspondence with friends in another, returned homework in a third, and so on. As with system folders, most mail systems allow hierarchies: folders of folders.

File Maintenance.

Just as a system can Delete or Print files, an Email program can Delete or Print messages. In general, most actions performed at the system level have analogs within a mail system, such as copying or moving a document, finding a document, or reordering documents based on different visualizations.

4.3.3

Email as a Word Processor

A message is usually a written document. Therefore, it is not surprising that virtually every Email system provides at least the rudimentary word processing tools. Usually only the basic tools (e.g., cut, paste) are provided. Fancier options (e.g., spell checkers or the ability to change the font and style of individual characters) are often missing.

Exercises

4.5 Compare your Email tool with the simple word processors you have used so far. For each word-processor command, try to find the analogous operation in your Email application.

4.6 Compare your Email tool with your operating system. For each file manipulation command, try to find the analogous operation in your Email application.

4.7 Introduce yourself to a classmate. Find out his or her user

name. Send a message to that classmate (and ask for a reply). Check to see that you get the reply.

4.8Send a message to your instructor (perhaps saying how much you are enjoying this course).

4.3.4

Hybrid Tools

The electronic nature of an Email system makes possible tools that have no direct analog in either the U.S. postal system or in file systems. In general, these tools do not solve problems for you, but reduce the tedious aspects so you can spend more time solving the real problems.

Phone Message System Tools.

Generally the post office does not take messages (in the sense of "Ms. Smith isn't here right now. May I. . ."), but a

phone answering machine does do that. Sometimes an Email system actually seems more like a phone answering machine than it does a postal service.

Reply.

When you receive a letter, you very often want to reply to it. Most Email systems provide an automatic *Reply* capability.

A Copy for your Records.

Most Email systems will keep copies of outgoing as well as incoming mail. This provides an excellent source of records or documentation.

Aliases or Nicknames.

Most users find that they contact certain persons repeatedly. Most systems allow such a user to establish *aliases* or *nicknames*. Thus, if you find that you are frequently sending messages to a user called Schnegglemeister, you could create an easy-to-remember alias for that user such as the first name: Al. Certainly, it will reduce typing errors if nothing else. Such an alias is just a *pointer* to the actual name.

Multiple Recipients.

One of the great advantages of electronic data is that creating copies requires no effort. This certainly applies to letters. To send a letter to two or more users, just include all of the addresses when you address the letter. In fact, you can usually create an alias that refers to both of the recipients.

4.3.5

Thinking About Mail Systems

Granularity of Operation.

A mail system works on letters or messagesentire messages. This

means that we should expect it to provide excellent services at the high-granularity level, but not expect the sort of low-granularity sophistication that a good word processor provides. Mail systems do provide some amount of low-level support, but for the most part, they are designed to help you send and receive messages. Most capabilities have similar granularity to the system itself.

Mail to Remote Locations.

You will see eventually (e.g., in *Chapter 22: Telecomputing*) that the mail system is not restricted to just your local computing system. Instead, it is part of a single international mailing system. This means that you can send mail wherever you want and to whomever you wish; the technique is essentially identical for sending a message across the room or around the world. If you are interested in sending mail to remote sites, do two things: first, look at Chapter 22 to find out about Internet addresses, and second, check with your instructor to make sure you have permission to do so.

Exercises

4.9 What else should a telephone answering machine do?

(table continued on next page)

(table continued from previous page)

4.10 Find out what privileges you have for sending mail to distant sites. If you know of someone at a distant site, send that person a message. (This is covered more completely in *Chapter 22: Telecomputing.*)

4.3.6

Suggestions for Using Email Systems

No two users will use an Email system in exactly the same manner. However, the following list of suggestions may prove helpful for making effective use of your Email system:

Use folders (directories) to reduce the “clutter” of your messages and to help you find the ones you need.

Develop a “throw away” policy. Get rid of those messages you do not need at all and file those that you do need.

Create a list of nicknames or aliases for those persons you contact regularly.

Check your Email on a regular basis (the speed of Email is wasted if you do not check it for a week).

Be aware of the Email habits of those you communicate with. If they don’t read their mail, they won’t see your message.

There are also several commonly accepted etiquette standards:

Most people are far more casual about spelling and grammar in Email than in other forms of communication but it still should be legible.

Remember that Email has few ways of showing emotion or emphasis. Be especially careful in messages that can be misconstrued.

Avoid using all upper case letters: it feels like you are shouting.

Be brief and to the point. Most persons do not want to read long letters, especially unsolicited ones. Remember that your message may be interrupting some other task that the person intended to perform.

Avoid sending messages to individuals who do not want to receive them.

4.4

Selecting the Best Tool

Users want to share information and documents for a variety of reasons. When should they use file servers and when should they use Email? Either tool allows one user to place information in the hands of another, but each has its relative strengths and weaknesses. These virtues can be understood best by considering the nature of the service that each provides: an Email system is a delivery device, but a file server is an archive.

A mail recipient needs to take very little action to receive a document: at the most, open the mail application and request that it read the mail. In particular, that user need know nothing about the specific message, its name, where it is stored, or even its existence.

Since Email tools inform the user of the presence of new mail, they are better in situations in which it is important to the sender that the receiver gets the information.

Email necessarily generates copies of information. A message with many recipients generates many copies. Thus, mailing large documents to multiple recipients can generate large quantities of duplicated data.

Email systems are primarily designed for text; their ability to handle other kinds of data is often somewhat limited. Thus, servers may be better devices for non-text oriented data.

Some data changes regularly. If all recipients know that the latest version will have a specific name on a specific server, they need only go to that place to get the data. Email would require that they receive every new copy as the data changes.

Recipients may not be as interested in the contents of a message as the sender. File servers allow users to get the material if they want it and ignore it if they do not want it.

When selecting a means of distribution, always ask about the needs and objectives of both sender and receiver.

Exercises

4.11 This section used electronic mail systems and file servers as an example for weighing the relative advantages of two competing tools. Perform a similar analysis to compare the relative advantages of electronic mail and phone answering systems.

4.5

Mail, Problem Solving, and Computing

Is mail a problem solving tool? Certainly it is a service that solves

problems such as getting a message there on time. Unfortunately, that is not the level of problem most people mean when they talk about “problem solving.” This is more like “solution selection.” Sending a message is not the sort of solution that you figure out. Sure, you must learn about the system, but that is really an issue of experience. You really can’t apply the learned knowledge to a totally new situation. Nonetheless, electronic mail fills some key roles in a problem-solving course.

Practice.

First, it provides practice on using basic computer skills: typing at the keyboard, selecting objects, finding help, and providing instructions to perform operations. Practice and repetition is a problem-solving skill (but notice that the problem solved here is not mailing messages). The more you practice this facility,

the more it becomes second nature and the less you will complain about “the computer getting in the way of the serious problem solving.”

Analogy.

Because Email works like so many other aspects of a computing system, its use practices that essential skill: understanding by analogy. In recognizing the similarities between applications, you better understand the applications and develop your skill for understanding future applications. Emphasizing the similarities rather than the differences also helps the user understand new applications.

Rapid Communication:

Email makes rapid communication possible. And rapid communication is a valuable commodity in an academic situation. For example, your instructor may only have a few office hours a week, but can always receive electronic mailsome even get Email at home. In the other direction, the instructor can send important information to the entire class quickly. For example, suppose a student asks a question about an assignment and the professor notices a typo on the assignment (one bad enough to cause confusion or misunderstanding). That instructor can send an Email message to the entire class informing them of the problem.

Reference Sources.

Asking questions is an essential problem-solving skill. In the library, the reference librarian provides useful information about sources of information. With Email, many additional sources are easily available for asking questions. You may ask questions of not only the instructor for your class, but any instructor you know who has Email. Many special services exist for the specific purpose of answering questions via Email. Learning to use these sources to your advantage takes practice. And most important, later chapters

will show ways to extend your Email question-answering horizons.

Tools of the Business World.

Finally, note that Email is becoming more and more common in the business world. On graduation, you will find that you are expected to communicate with your colleagues electronically. If you get the experience now, you will be well ahead in the end.

4.6

Questions as Problem-Solving Tools

One way to solve a problem is to ask a question. In fact, this is often the best way to solve a problem. In particular, any situation in which solving the problem yourself constitutes “reinventing the wheel” is a good candidate. Why solve a problem someone else has just solved? Another important group of problems are those requiring information that only an expert is likely to possess. For example, if you thought you might have appendicitis you would consult an expert. Electronic mail brings additional expertise closer.

4.6.1

The Question as Teamwork

Asking a question establishes a team approach to problem solving. Admittedly, it is a very one-sided team effort, but both participants do bring information to the discussion. One brings expertise about a general problem area. The questioner brings specific information about the problem (e.g., the appendicitis victim in the previous paragraph brings an intimate knowledge of the symptoms she is experiencing). This information is crucial to the problem-solving process. No amount of expertise can solve a problem without knowledge of the details of the specific problem.

Bring Specific Knowledge to the Table.

You would not say to a physician, “Fix my pain,” without describing the pain, or to the mechanic, “Fix my car,” without saying what is wrong with the car. So, when you experience computational problems, bring the symptoms to whomever you ask for help. Do not say, “It doesn’t work.” That tells very little. Explain what went wrong as precisely as you can. Types of information that are useful to the experienced user include:

What was the error message? (The exact error message. Copy it word for word.)

What were you doing at the time the problem occurred? (What application were you using? What operation were you performing? What had you done immediately before?)

Is the problem repeatable? (Does this situation occur every time you perform the same actions? Yes, this does imply that you should try it again.)

Is there any other information that seems directly relevant to the error message? (If the error message says “no space on the disk,” find out how much space is on the disk; if it says “cannot find

application,” can *you* find the application?)

Often you will find that the simple act of preparing to ask a question will itself provide the answer. Even putting the question into words helps. Formulating a question carefully forces you to think through the details to a level that you might not otherwise do.

4.6.2

Extending the Concept of Communication

Chapter 22: Telecomputing discusses wide area nets. Even though such nets are not discussed until later, considering some implications of nets now is both interesting and thought provoking. What would be the implications of a net that covers not just an academic department, but an entire country? Certainly electronic mail could then be sent to the outside world. Files could also be made available to very large numbers of people. On the other hand, addressing mail to outsiders must be slightly more complex than addressing it to insiders. Similarly, accessing large file servers is slightly more complex. However, the basic principles are the same.

4.7

Privacy Issues of Electronic Mail

Gentlemen do not read each other's mail.

HENRY L. STIMSON

The analogies between traditional and electronic mail extend well beyond the basic applications. Perhaps it is not too surprising that computerized junk mail already exists, but two other developments are slightly more unexpected:

electronic harassment and

individual privacy rights.

Each of these three problems merits some discussion.

4.7.1

Junk Mail

The bulk of the contents of most postal mailboxes on any given day seems to be unsolicited ads. While most people complain, they seem to accept it. The immediate solution5 is relatively simple: throw it away (often unopened). In addition, the post office claims that their receipts from junk mail help keep down the costs for the rest of us.

Computerized junk mail raises a slightly different problem. One user can often send a message to every user in a computer network with the same amount of effort as it takes to send it to a single user. In some systems the cost may even be the same. In addition, computerized junk mail may be a little more difficult to ignore completely. For example, if your system beeps at you whenever you receive new mail, you will probably check the mail only to see that it is junk.

Many institutions have policies against using their facilities for large-scale distribution of unsolicited mail. Users should respect the

policies of their institutions. Most institutions operate on (at least semi-) democratic principles. If users do not like the policies, there is usually a mechanism to change them. But if you want to distribute junk mail, do not expect much sympathy from your colleagues.

4.7.2

Electronic Harassment

Harassment is more commonly associated with the telephone than with paper mail. The latter is largely confined to such formal actions as billing disputes. Telephone harassment also includes more obnoxious forms such as sexual harassment, vulgarity, and requests for sexual favors. Oddly enough, there is a small contingent of computer users who seem to gain pleasure from the equivalent computerized harassment.

There isn't even an interesting ethical issue to debate here. An analogy to telephone harassment seems to resolve virtually all questions on this issue. If you

5. This obviously does not address any environmental issues. Hmm, another set of problems!

would not do this on the telephone, do not do it using a computer. It is equally inappropriate.

The existence of electronic harassment is indeed odd for two reasons. First, unlike telephone calls, Email is easily traceable. A person who receives harassing telephone calls must request special action from the phone company. The perpetrator must then harass a second time to be caught. No special action is required to “put a trace” on electronic messages. Messages come with all the information needed for an electronic trace. Computing center officials at the institution can usually pinpoint the perpetrator with ease.

In addition, penalties are easily imposed. In an academic institution, the first and obvious penalty is denial of computing privilegesa hefty penalty indeed if you need the computer to complete your homework. While the appropriateness of specific penalties and the mechanism for imposing those penalties can be debated, few would debate that some penalty is appropriate. Thus, electronic harassment is conducted in an environment of easy detection and direct penalties.

4.7.3

Privacy of Communication

If Person A receives a traditional letter in the mail, Person B would seldom open it without explicit permission of Person A. Nor would B read the letter without permission. It has been more than half a century since Stimson made his comments (see the beginning of Section 4.7), but they apply just as well today to computer mail. A computer screen is more difficult to keep private than a sheet of paper, but that does not change the guideline. The fact that it is relatively easy to snoop does not make it more appropriate. Electronic mail is the private property of the sender and receiver.

Two more interesting questions do arise. First, does the owner of

the computing system (e.g., the university) have the right to check the mail of individual users? On one side, users maintain that electronic mail should be equally as private as the U.S. mail (whose privacy is guaranteed by law). On the other side, computer service providers may argue that the mail was delivered using their equipment. That is, they pay the bills, so they have the right to control how the machines are used. Even users who own their own computers use the services of the “post office” provided by the institution. Employers may take the argument a step further: they may claim they have a right to monitor the work done in their name. They may want to see that employee use is indeed appropriate, and that employees are not using company facilities for private gain or to send company secrets to a competitor.

Exercises

4.12 Create a list of situations when “junk mail” might be acceptable. Select one and make an argument supporting such use.

(table continued on next page)

(table continued from previous page)

- 4.13 Write an essay on freedom of expression (as guaranteed by the U.S. Constitution) versus freedom from harassment.
- 4.14 Take a position on the right of an employer to read employee electronic mail. Defend your position in a short essay. Note that numerous articles have been written on this subject. You will do well to read one or more before writing.

4.8

Summary

Data may be shared either by placing it on publicly accessible disks or by using electronic mail. The former makes arbitrary information available to large groups of people. The latter most commonly makes text available to a small group of selected individuals. Sharing data opens new problems such as privacy issues and susceptibility to viruses.

Important Ideas

file electronic client
server mail

virus harassment network

5

Representing Information

Some problems are so complex that you have to be highly intelligent and well informed just to be undecided about them.

LAURENCE J. PETER

Pointers

Gateway Ad hoc Organization

Lab: Techniques

Lab Unit 4: Separating the Problem from

Manual: the Environment

5.0

Overview

An understanding of the representation of information particularly textual representations provides insight into possible solutions for two separate problems. First, such insight helps you design written documents that better deliver the information you wish to communicate. Proper use of organizational techniques makes your documents more comprehensible. Second, it provides a basis for understanding potentially confusing behavior of user software, helping you select the correct actions to achieve the results you want.

5.1

Three Views of the Structure of Textual Information

The test of a first-rate intelligence is the ability to hold two opposed ideas at the same time, and still retain the ability to function.

F. SCOTT FITZGERALD (*The Crack-Up Esquire*, February 1936)

Information is the “knowledge derived from a study” or the

“communication of knowledge.” The act of writing is clearly the latter. Successfully communicating information requires organizing the information, and organizing requires some understanding of the representation of that information. This does not mean the internal computer representation of data. It is not necessary to understand that any more than it is necessary to understand how your larynx represents sounds as vibrations or how a fountain pen draws ink through capillary action. But it is

necessary to understand the structure or organization of the information. The word "cat" is a written representation of the concept of a soft furry animal that likes to curl up in your lap. "Purr," and (as the French say) "*ronronne*" are representations of the sound that the animal makes. The corresponding spoken word is yet another representation. It is necessary to understand the links between these concepts but not necessary to know the internal representation of that word in a computer.¹

The simplest form of information is simply a "yes" or "no" answer.² But individual pieces of data are seldom completely isolated. Almost all information is grouped using some grosser structure. Even a "yes/no" answer needs to be associated with a question. And, even in simple English prose, the letters in a collection of text are grouped into words, sentences, paragraphs, and even chapters.

Language in One Dimension.

The simplest way to view textual information is as one long *stream* or *string* of characters. Each character (except those at the extreme ends) is preceded and followed by other characters. Each has exactly one next character and exactly one previous character. The words *sequential*, *stream*, *string*, and *linear* all describe this property of written information. The activities of reading and writing both treat text as sequential. (Although one could argue that the basic unit is a word, not a character, the underlying principle is the same.) A typewriter is oblivious to further structure. Similarly, many word-processor operations are sequential in nature: insertion of text, searches for key words, and spelling correction all work sequentially.

Language in Two Dimensions.

Even though language is essentially linear, written documents also seem to be two-dimensional. The page of a book is inherently two-

dimensional. For example, Figure 5.1 shows two views of a sentence: a one- and a two-dimensional view respectively. In the first, the notion of “next” is clear; in the latter, there could be some confusion: next in the same line, or next below? Does “next” even apply to the last word of a given line?

5.1.1

More Complicated Structures

Hierarchical Language Views.

Text also has an even more complicated structure incorporating multiple layers. Characters are grouped into words; words into sentences; and sentences into chapters or sections of an entire document. By

1. For those who really care “cat” is stored internally as 011000110110000101110100. Representations such as this are covered in courses in computer organization, or discrete mathematics.
2. The information content of a “yes/no” question is called a *bit*, short for *binary digit*. Binary arithmetic has only two symbols (0 and 1). Two symbols are exactly enough to represent any pair of values, such as “yes” or “no.”

Man is still the most extraordinary computer of all (John F. Kennedy).

Man is still the
most
extraordinary
computer of all
(John F.
Kennedy).

Figure 5.1
Two Views of One Sentence

accepted convention in written text, blank spaces separate words, periods separate sentences, and new lines separate paragraphs. People group the characters into words when reading aloud. They may even add markers inflection, volume, pauses to reflect the sentence structure. People seldom stop reading in the middle of a paragraph.

Large written works need to be segmented into a series of sections, such as chapters in a book. Each segmentation creates two levels of linear structure. Chapters have an implicit order, and the text within each chapter is a linear sequence. The visual segmentation normally implies an underlying logical segmentation. That is, each chapter of a book should be a logically coherent unit. A chapter should be visually separable from other chapters. The next chapter of this text addresses more formal hierarchical organizations for information.

Irregular Structure.

Importance, sarcasm, and so on, are irregular and usually require more complicated representations. In spoken language the speaker indicates these aspects by using inflection or stress on certain words. There is no word that means “this is a question” the role filled by a question mark. Instead, “You went to the store?” is often spoken with a rising note at the end of the sentence to indicate that it is a question and not a statement of fact. There is no single technique for representing the nonlinear properties in written material. For example, at the sentence or word level, a writer may

show emphasis or structure through the use of font changes such as bold or italics. Section 5.3 begins a discussion of such techniques.

5.1.2

Dealing with Multiple Views

Working with two or three distinct models is not necessarily a problem. For example, most people use two quite contradictory models when thinking about the planet on which we all live. Depending on which model they are using, Earth is either flat or round. We know that it is “really” round and that if we travel in one

direction, say west, we will eventually get back to where we started. In contrast, words like *level* describe anything parallel to the ground (parallel to a curved surface?) and *flat* describes bodies of water. For the vast majority of our day-to-day activities most people behave as if the Earth were indeed flat. Most maps represent the earth as flat. This seldom, if ever, causes any problems. Multiple models of computers and information provide similar advantages, but the user needs to understand the difference.

5.2

Information Structure as a Problem-Solving Tool

Although text viewed as a single long sequence of characters provides a simple computational model, it is not particularly useful to the human reader. People certainly expect to read words, sentences, and paragraphs. Readers often use their expected structure to help fill in the needed organization, but, just as frequently, they want even more structure explicitly indicated. Without additional structure, they may fail to understand some aspects of a text. Even with apparently well-organized material, the reader may fail to make the necessary connections for any number of reasons:

Some information cannot be represented linearly in any nice way.

Humans are frequently in a hurry.

The reader may not care as much as the writer does about the material.

Humans do not always perform at their best.

Fortunately, it is possible to organize a written document so that its structure reinforces the underlying structure of the material being presented. The structure helps the writer deliver points coherently and helps the reader understand those points.

5.2.1

Whose Problem is it, Anyway?

If a reader does not understand material because she is in a hurry, whose problem is it? Some writers might try to dismiss the hurried-reader problem with:

*If she wants to miss information by reading too fast
that's not my problem.*

One could counter this philosophy with a classic proverb: “do unto others. . .,” but there are more fundamental problem-solving reasons for the writer to address this problem.

Writing is Communication.

The goal of writing is to communicate information. If the reader misses the point, the writer’s goal is also missed. Most writing has a purpose, and generally that purpose is important to the writer. If the reader does not understand part of the content, the writer loses. The writer wants to do everything possible to make the writing understandable. Suppose you are

Box 5.1 Documents That Must Be Really Legible

Funding agencies are government or private organizations that give money to colleges to develop new ideas. Faculty who hope to receive a grant submit detailed applications, which may be 25 to 50 or more pages long. Funding agencies frequently reject the majority of applications on the basis of a first readinga reading that takes less than half an hour. The application that is not impeccably organized will not get its point across.

Job candidates are often invited for an interview on the basis of a very short preliminary scan of the information in a résumé. Applicants with poorly written résumés do not get interviews. Without an interview, they have no chance to demonstrate their other skills.

Even the preliminary categorization of college applicants may be based on a very cursory review of the material. Suppose that Grateful-Dead University receives 10,000 applications per year. All applicants submit their applications by the January 15 “Dead” line and the college sends out its acceptances or rejections 12 weeks laterby April 15. The admissions office breaks the process into the following steps:

- gather material (2 weeks)
- preliminary screening
- final selection (2 weeks) and
- notification (2 weeks)

That leaves six 40-hour weeks for the 10 staffers to read the 10,000 applications. That is 2400 hoursor less than 15 minutes per applicant (and that assumes no coffee breaks, no sick days, holidays, or time spent doing any other work activities).

Conclusion: If you do not make a great first impression in 15 minutes, you don’t get into the college. Organize your work well.

writing an essay entitled “Why I should get an Alfred E. Neuman Fellowship.” It is in your interest to make sure that the reader reads and understands every one of your major points. Yet, the reader may be reading 1000 such essays. She will therefore be somewhat rushed. It is the writer’s responsibility or at least to the

writer's advantage to ensure that even a careless or rushed reader will see and follow the major points.

Exercises

5.1 Professor Smith has 123 students in class and gives a midterm containing eight problems. She decides to devote a single weekend to the task of grading the exam. Assume that she wants to start working no earlier than 8:00 A.M., stop no later than 5:00 P.M., and take a one-hour lunch, and two 15-minute breaks each day. On the average, how much time can she devote to one answer for one problem from one student?

5.2 Discuss the implications of Exercise 5.1.

5.3

Visual Tools for Emphasizing Structure

5.3.1

Ad Hoc Techniques for Organization

Ad hoc means “created for the specific situation.” An *ad hoc* solution to text organization forces an apparent organization on text by brute force means. For example, bold words stand out, indicating primary importance. Similarly, large or small characters, underlined or *italicized* text, changes in font all imply structure. Each tool can be used in several ways. For example, a writer may

shout some information!

Unfortunately, too many writers employ *ad hoc* structuring techniques without first imposing a logical structure on the document. No visual or *ad hoc* structuring tool can make up for a lack of logical organization in a document. *Chapter 6: Hierarchical Organization of Information* addresses important tools for creating logical organization.

Over the centuries, writers have established several conventions for the use of visual appearance to emphasize both *ad hoc* and systematic structure in documents. Basic building blocks for showing structure include: fonts, styles, sizes, and graphics.

5.3.2

Fonts

Font refers to the general appearance of the type face:

1. Computer science is fun.
2. Computer science is fun.
3. Computer science is fun.

are three representations of the same sentence. They are the same size and style, but are distinct fonts. Fonts help establish the general feel of a document. Some

fonts feel very formal. Others feel more casual (e.g., number 3 above). Some are reminiscent of special situations. One relatively modern convention is the use of a *sans-serif* font (see Section 5.6.1) to indicate computer input or output (as in number 2 above). Changes in font indicate special interpretations. Since most documents use a single font throughout, any such change signals to the reader that something special is occurring. (See the Preface for the use of fonts in this text.)

5.3.3

Styles

Styles are applied to the basic typeface to change the appearance, while keeping the same general shape of the characters. Usually documents are written almost entirely in a single style. A change in style indicates a special use or emphasis. Some common styles and their uses include:

Italics. Italics can indicate (a) that the given text is not in English but in another language (frequently, Latin, as in *ad hoc*); (b) the titles of books, magazines, or journals³ (as in *Problem Solving with Computers*); or (c) the introduction of a new term (e.g., *font*).

Underlines and bold face. Underlining and bold face type are used to indicate important concepts. Generally, the concurrent use of both should be reserved for special situations, such as headers, or sentences in which it is very important to stress a single word.

Special-effect fonts. Special fonts such as shadow-letters and outlines should be reserved for the use implied by their name: special situations. The **shadow** font may make a good attention grabbing title for a poster. Use them very sparingly: they tend to make text harder to read. These fonts should almost never appear in a formal document.

Strike-through letters. Strike-through letters are used in legal documents to indicate changes in the text. Both old and new

versions appear in a single document with the characters of the old version struck-through as if they have been crossed out. This enables the reader to compare the old and new versions of the document easily by looking at only the single document.

3. Note: Until relatively recently, most typewriters could not produce italic fonts. Therefore, editors adopted an alternate convention: authors could underline text in manuscripts to indicate sections that should be italicized in the final version. This convention was also adopted for typed works that were not intended to be typeset (e.g., term papers). With the advent of word processors, this convention is no longer needed. Word-processed documents should not use it.

Although fonts are mutually exclusive, styles are not. That is, style describes a transformation of the basic font. An instance of a character can be both underlined and italicized (*example*). But it can be only one font.

5.3.4

Size

Larger and smaller characters make sections stand out more or less, respectively. Size is measured in *points*, a term borrowed from the printing industry. Each point is approximately 1/72 of an inch.

Thus, 6 rows of 12-point type could fit in one inch. Any character can be any one size, as demonstrated in Figure 5.2.

5.3.5

Additional Structuring Tools

In addition to structuring blocks of individual characters, logical structure can be reflected through the visual representation of larger blocks such as paragraphs.

Paragraph Shape.

The size or style of paragraphs can be varied. For example, text indented on both sides often indicates an extended quotation. The line spacing may be single or double spaced. Use or non-use of a first-line indentation on paragraphs can affect the feel of a document.

Reference Tools.

The table of contents, indexes, and cross-references all illuminate the overall structure above and beyond the regular sentence-paragraph structure. They are discussed in more detail in *Chapter 21: Graphs and Hypermedia*.

Pagination.

Documents gain structure through the appearance of individual

pages. In addition to numbering that shows order, pages can indicate segmentation (e.g., the chapter name or number) or continuity (e.g., the author and title).

9 points 12 points 18 points 24 points

72 points

Figure 5.2
Self-Descriptive Examples of Font Sizes

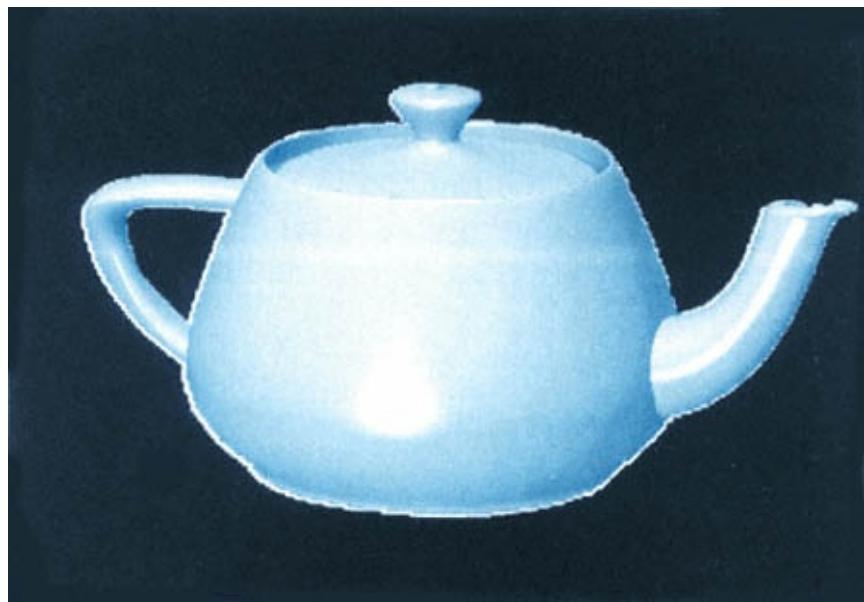


Figure 5.3
The Classic Graphic Teapot

Source: This little teapot may be the most frequently drawn piece of houseware in the world. It serves as a common exercise for advanced computer graphics. This particular version was created by Brad Grantham of Silicon Graphics.

A picture is worth a thousand words, and graphic material such as Figure 5.3 can be included in text. See *Chapter 7: Models, Visualization, and Pictures*.

Exercises

5.3 Many word processors have separate menus for font, style, and size. Others combine these into a single menu. Write a short sentence with your word processor. Copy and paste the sentence repeatedly, using as many combinations of font, size, and style as you can.

5.4 Does your word processor include the fonts Helvetica and Geneva? If it does, compare them. Comment on any similarities and their names. (Hint: look up Helvetica.) Can you find any other such pairs of fonts?

5.4

Selecting Tools

No absolute or universal rules exist for emphasizing the logical organization through visual tools. For example, both style (italics, bold) and size (larger) are

used for emphasis. However, several observations and rules of thumb can help the new writer get started:

Use techniques that you have seen in published work rather than inventing new ones. The reader may not understand a new and unexplained convention. But use of a familiar convention reinforces the reader's understanding.

Use bold or larger characters to mark the beginning of sections.

Use bullets (as in this segment) for lists or series of important items that are all similar in some basic way.

Use numbers instead of bullets if you wish to refer back to the items in later text.

Use italics to indicate words that are newly defined (as far as the reader is concerned) or that are used in special ways.

Use indentation and headers (entire lines in a larger font) to indicate or delineate sections of items that are grouped together.

Use nonalphabetic (special) characters to provide other special meanings.

Indicate the difference between computer terms and noncomputer terms.

On the other hand, too many different techniques and structures make the document confusing. Fortunately, the next chapter provides more systematic methods for creating structure. The use of computerized tools does not remove the need for appropriate style. Always use a style guide such as the two mentioned at the end of the chapter.

5.5

Structure and Computerized Documents

The common logical and physical views of text also create conflict

for a word processor. Logically, text can be one long string of charactersa one-dimensional structure. But on a printed pageor a *wysiwyg* screentext is a two-dimensional structure. Word processors do in fact maintain both models or metaphors: behaving in some situations as if text is linear and in others as if it is two-dimensional. The interactions of the two models can sometimes confuse users.

5.5.1

Wrap Around

A printed page is a two-dimensional object. Even a relatively short sentence can exceed the width of a page. Obviously, the printed representation of a sentence may require two lines. We say that the sentence *wraps around* from the first line onto the second. Put another way, the logically one-dimensional string of characters is also a two-dimensional physical structure. Adding or deleting characters assumes a linear representation. If the user adds characters in the middle of a paragraph (changes the logical structure), all subsequent characters must move logically to the right. Characters that do not fit on the line, wrap around. The entire paragraph may change shape (in two dimensions) to accommodate the new characters.

Similarly, the number of characters per line is dictated by the page and font sizes. Changing the size of either the page or the characters changes the shape of the paragraph. For example, repeating the paragraph on a narrower page creates:

Similarly, the number of characters per line is dictated by the page and font sizes. Changing the size of either the page or the characters changes the shape of the paragraph. For example, repeating the paragraph with a smaller font creates:

Similarly, the number of characters per line is dictated by the page and font sizes. Changing the size of either the page or the characters changes the shape of the paragraph. For example, repeating the paragraph with . . . creates:⁴

5.5.2

Selection

Recall that the description of any action or operation requires both an operation and an operand. For manipulating text, the user must select the text to change as well as the change to make (see Section 1.2.1). In the case of existing text, it is easy: select the text and then describe the change, as in Figure 5.4 on page 82.

- Now consider the interaction of the two views of text. Suppose you wish to select a paragraph. Because the paragraph is displayed on a two-dimensional screen, the selected area is roughly rectangular. Because data is also viewed linearly, it can be selected by selecting its beginning and end points (marked here by the two cursors).

Conversely, although it might be easy to visualize a vertical rectangle that intersects multiple lines within a paragraph, that rectangle has no simple analog in the linear representation. For example, the rectangle intersecting this paragraph does not correspond to any single segment in the linear representation of the paragraph. Therefore, there is no nice way to select the text bounded by the rectangle.

5.5.3

Invisible Characters

Gutenberg's invention of movable type five hundred years ago required one simple observation: "if all characters are essentially equal, their positions can be freely interchanged." This observation applies to punctuation marks as well as the letters and digits and the *blank character* or *space*. For at least five hundred years,

4. Even with the minor wording change of the second repetition of the paragraph, each repetition seems to imply that there will be yet another paragraph. I prudently chose to stop after three. If I were to add a fourth, it would seem to require a fifth, which in turn would require a sixth, and so on. Popularly, this is known as an *infinite regression* a series which never stops. Computer scientists call this *recursion*. Properly controlled, it can be a powerful tool as we will see in *Chapter 24: Algorithms revisited*.

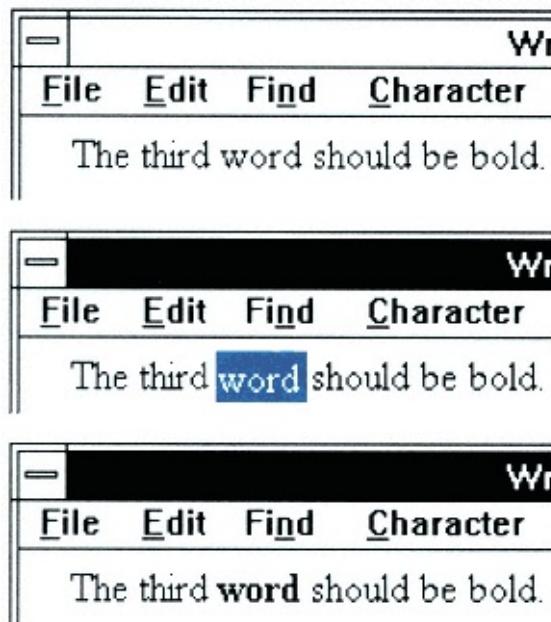


Figure 5.4
Two Steps for Specifying a Change

bookmakers have treated nonalphabetic characters as characters, with blank characters separating words. A blank die can occupy an empty space. That is, in a printing press, a blank is as tangible an object as any other character. Word processors have inherited this as the primary tool for organizing simple character data. In particular, *tabs* and *carriage returns*⁵ are characters in the linear stream of data. Typing these keys places the *tab character* or *carriage return character* into the stream of characters. This character is not visible⁶ but behaves like any other character in all other ways. It can be selected, deleted, copied, moved, or replaced.

5.5.4

Null or Empty Objects

'Just look along the road and tell me if you can see either of them.' 'I see nobody on the road,' said Alice.

5. The name *carriage return* is an anachronism derived from an era when typewriter carriages returned to the right when the user pressed that key. Most computers now label that key just *return* or *enter*, but a better name would be “end of line” or “end of paragraph.”

6. Many word processors do allow users to request that the application display visible representations for these invisible characters on the computer screen. The user may turn the feature on or off, but the characters will always be invisible on a printed document.

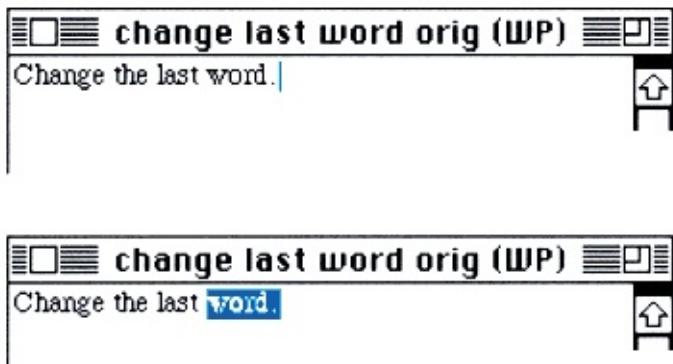


Figure 5.5
Two Situations for Delete

'I only wish I had such eyes,' the king remarked in a fretful tone. 'To be able to see nobody, and at that distance too! Why it's as much as I can do to see real people, by this light.'

LEWIS CARROLL (*Through the Looking Glass*)

Suppose a user wants to change the last word of the text in Figure 5.5. In the first case, the selection point is indicated by the cursor, and Delete or Cut works just like a typewriter by backspacing. In the second case, a segment is already selected, and at first glance it may appear to behave differently. In reality, both situations behave in the same manner: the new material replaces the selected material. But, wait! There isn't any material selected in the first case. True, but computer scientists prefer (like the king) to say no characters have been selected. The selected range is *null* or *empty*. Figure 5.6 on page 84 shows two more examples. The first section contains four rectangles, each shorter than the one above. The last one is so short, it has no width. We say it has a width of zero. The second section shows a related text problem: four strings of a's, each sandwiched between a pair of b's. The last string is so short, it has no characters; it is null. In each case, a command "replace the string of a's with an x," has exactly the same result: $b \times b$. Examples of null objects are common in computer science. Look for them. They help impose a consistency on operations. You could even say that a document is a "null document" before you write anything on it.

5.6

Describing Text and Characters

Fonts can be grouped according to their properties or characteristics. This categorization provides a vocabulary for describing characters. The two most commonly referenced properties are *serifs* and *spacing*.

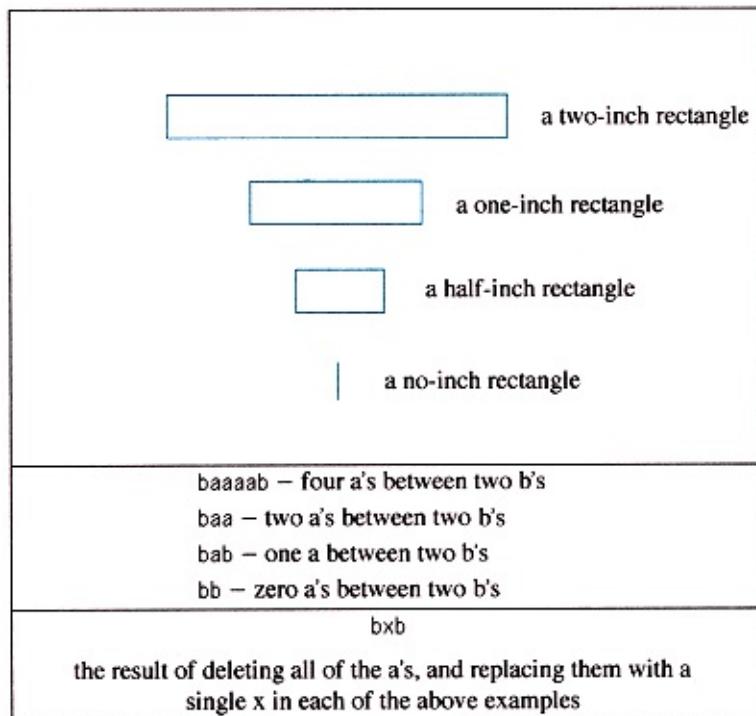
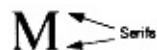


Figure 5.6
Null or Empty Objects

5.6.1 *Serifs*

Serifs are small lines or extensions at the extremities of the character:



A font with serifs is called a *serif font*. Fonts with no serifs are called *sans serif*. (Courier, New York, and Times are all *serif* fonts; Geneva, Chicago, and Monaco are all *sans serif*.) Most people find serif fonts easier to read than sans serif fonts (but there is some evidence that the opposite is true on computer screens).

5.6.2 *Spacing*

Fonts may be either monospaced or proportionally spaced. In a

monospace font, every letter requires the same horizontal space. An “I” gets as much space as a “W”. In *proportionally spaced* fonts, the amount of space allowed to each letter

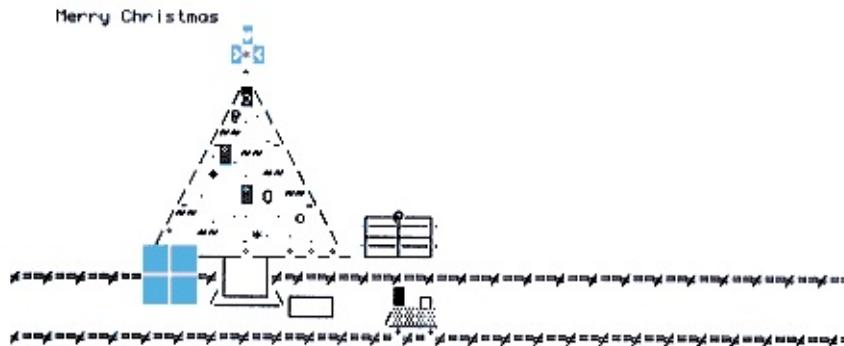


Figure 5.7
Picture Made of Printer Characters

Source: This card has been widely distributed on the Internet for years.

Unfortunately, the identity of the original author has long since been lost. The original is more ingenious: the train actually appears to move along the tra

is based on the complexity of the letter. Since an "I" has only one stroke, it requires less space than do the four strokes on a "W". Although most readers find proportionally spaced fonts more visually pleasing, monospace fonts are historically important for two reasons early typewriters, the carriage advance mechanism necessarily moves the same distance for each character. Subsequently, the analogous problem made monospace fonts more practical in early computer printers and monitors. If every character requires the same amount of space, the position of the second character is constant independent of the particular first character.

Computer users have often taken advantage of this property of monospaced fonts to create pictures or special effects using the standard typed characters, such as the Christmas card in Figure 5.7. Today such text pictures are much less important. They have largely been replaced by much more powerful computerized drawing tools (*Chapter 7: Models, Visualization, and Pictures*). However, they are used in situations where actual pictures may not be practical. For example, computer mail between machines of different types may not support drawn pictures (see *Chapter 3: Documents as Objects*). Even though dated, many texts today continue to use monospaced fonts to indicate computer input or output.

Exercises

- 5.5 For eight of the fonts available on your machine, classify them into one of four groups based on the presence or absence of serifs and the spacing.
- 5.6 Suppose you were not sure if a given font were monospaced or proportionally spaced. Devise an experiment to determine which form it has.

(table continued on next page)

(table continued from previous page)

5.7 Use a monospaced font to write your name in letters at least 6 times the size of the font.

5.6.3

Ascenders and Descenders

Individual letters may be described in terms of the ascenders and descenders on lower case lines. A *descender* is a portion of the character that descends lower than the bottom of most characters. When writing on lined paper, they are those portions which fall below the line. Lower case letters

g, j, y, p, q

generally have descenders in printed characters. In cursive writing, f and z also are generally written with descenders. *Ascenders* are the portions of lower case letters that ascend or extend above the line formed by the tops of most lower case letters. Typically these include

b, d, f, h, i, j, k, l, and t.

Since all upper case letters are the same height, the terms ascender and descender normally apply only to lower case letters. Some computer fonts (usually those used in older printers and monitors) do not have descenders. Some (often the same ones) create lower case letters as smaller versions of the upper case letters. These also will not have ascenders.

5.6.4

Special Fonts and Special Characters

Writers often find that they have a need for special characters for characters that are not on the standard typewriter keyboard. This problem is especially acute for scientists and mathematicians who

may need to write things like

$$\int_a^b x^2 dx \quad (\text{EQ 5.1})$$

or

$$\forall x, y, z (x \subseteq y \wedge y \subseteq z \Rightarrow x \subseteq z) \quad (\text{EQ 5.2})$$

Prior to word processors, these symbols were often handwritten into an otherwise nicely typed document. Some secretaries had special “keys” they could manually hold against the typewriter ribbon. They then struck the key with a regular typewriter key, transferring the needed symbol. Special symbols are defined in two ways. Some are extensions of a standard font. In the same way that a shift key determines if an upper or lower case letter is typed, an *option* or *alternate* mode key specifies an alternate of a special character. In addition, entire fonts of special symbols are available on most machines. Some contain scientific symbols such as the one in (EQ 5.2). Others contain small icons for creating pictures.

Exercises

- 5.8 The most common font for representing scientific symbols is called `Symbol`. Find the `Symbol` font using your word processor and use it to write the mathematical expression in (EQ 5.2).
- 5.9 Most computing systems support pseudofonts such as `Dingbats` or `Wingdings`. These are not really traditional fonts for representing letters, but are special shapes such as icons or arrows. See if your system supports one of these fonts. If it does, create a chart showing for each letter of the alphabet, the corresponding special character in that font (for example, using the `Symbol` font, a becomes a; b becomes b).
- 5.10 Superscripts and subscripts are generally considered to be styles rather than separate fonts. Use a single font, to write the quadratic equation ($a^2 = b^2 + c^2$). Try to find a way to write it in its more explicit form:

$$a = \sqrt{b^2 + c^2}$$

by adding a change in font.

- 5.11 Use the `Symbol` font to help write some polynomial of your choosing. Now write both the derivative and the integral of that polynomial.

5.7

Syntax and Semantics of Natural Language⁷

Every language has both *syntax*the allowable symbols and structures of symbols and *semantics*the meaningful statements of the language. For example,

Colorless green ideas sleep furiously.

is often cited as an example of a syntactically correct but semantically meaningless sentence. In contrast,

I ain't got none.

is a meaningful, but syntactically incorrect, sentence. Both syntax and semantics are essential for good writing. Many computer systems provide tools to aid authors with both aspects, but not surprisingly, they do better with syntactic issues, which tend to be very mechanical. Semantic questions often require more judgment. For the moment at least, humans are much better at recognizing semantic problems.

7. A *natural language* is any language that humans naturally speak or write, such as English, German, Sanskrit, or Hindi. Computer scientists use this term to distinguish such languages from formal languages, such as those used for communicating with computers. Interpreting formal languages is actually a much simpler task than interpreting natural language. The latter task is a major component of the discipline of *artificial intelligence*.

Word processors can recognize English words and help correct spelling. Many provide an online *thesaurus* to help users select the correct word or the best word for a context (none can recognize the use of an incorrect word). Some can even detect errors in the structure of English sentences. Use these tools if they are available to you.

Blind Use of Tools.

There is no longer any excuse for computerized documents containing simple spelling errors. But, you do need to exercise caution when using tools such as spelling checkers. Recognize their limitations. A spelling checker is just that: it checks the spelling of a word to make sure that it really is a word. It can not tell if it is the correct word. It is up to the user to verify that the sentence is meaningful:

Their air know one butt ewe hew no how eye can due that.

5.8

Plagiarism Made Easy

Computerized tools not only make copying of applications easy, they make copying of text or selections of text just as easy.

Example 5.1 Consider the following scenario:

Maria took a course entitled *The History of World Civilization*. She earned an “A” for her term paper, “The Greeks and the Romans.” Her sister Frederica, a student at a nearby college, “borrowed” the term paper, and made a few changes using white-out. In particular, she changed the name of the author; the title became “The Romans and the Greeks”; and a few additional words were changed here and there. She then submitted it, as if it were her own work, as a paper for her *Cross-Cultural Interactions* class.

Few members of the academic community faculty or

student would describe her conduct as honest. Almost all would agree that some penalty is in order. Side issues, such as

Perhaps she did not explicitly say it was her work (she just submitted it as if it were).

It was not submitted to the same instructor or same school.

Perhaps she actually retyped the entire paper, changing sentences freely as she went.

8. Yes, this was intentional.

should have little impact on the basic question. She presented the work of another as her own. Period.

In contrast, Frederic borrowed and copied a computer file containing his sister Mary's term paper and modified it with a word processor. Some students think this is a different situation. The issue is exactly the same. It is certainly easier to create a new version. No tell-tale white-out marks give him away. The title, name, date, class, etc., are easily changed. Words can be changed throughout the paper with global find-and-replace actions. Even the format can be changed to make it look different to the eye: the page width can be made wider; headers made bold; whole paragraphs or figures moved to other places in the document. There is no question: this is a plagiarized work. Period.

Exercises

- 5.12 Find two reasons that people may use to rationalize their decision to plagiarize using a computer.
- 5.13 Try to find an example that is borderline: hard to tell if it is ethical or not. Write a paragraph defending each position.

5.9

References

Two widely accepted writing style guides are:

Strunk, W. & E.B. White. *The elements of style*. New York: MacMillan, 1979.

Turabian, K.L. *A manual for writers*. Chicago: University of Chicago Press, 1973.

5.10

Summary

Information has structure. Computerized tools help authors represent that structure within the constraints of a two-dimensional page, by modifying the font, size, and style of individual characters or words, and by the formatting of paragraphs or pages.

5.10.1

Actions on Parts of a File

inserting words in text

changing words

deleting words

changing the way words look, etc.

Important Ideas

document font syntax

application style semantics

disk size natural
language

plagiarism null

6

Hierarchical Organization of Information

I think that I shall never see A poem as lovely as a tree.

ALFRED JOYCE KILMER ("Trees," *Poetry Magazine*, August 1913)

Pointers

Gateway *Trees*

Labs: *Outliners*

Lab Unit 5: Hierarchies as an

Manual: Organizational Tool

6.0

Overview

A collection of data or information often has a natural or inherent structure. Readers will more easily understand the written representation of that information if it models the inherent or underlying structure. Unfortunately, the *ad hoc* organizational techniques discussed in previous chapters seldom capture the underlying structure. One of the most common basic structures is the hierarchical organization, the tree. Computer scientists love trees, but not for the same reasons as Kilmer. Tree structures provide a common organization for data throughout information and computer science as well as most other fields. The common outline is perhaps the most familiar tree structure, and many computing systems provide outline tools that assist users in applying hierarchical structures to text documents.

6.1

Problems in Data Organization

Unfortunately, treating information as a long, one-dimensional

stream of data or series of characterseven with *ad hoc* structure superimposed on that stream (as

suggested by Chapter 5) causes problems for both the creator and the user of that data. Unless they use the data in a strictly sequential manner, users will have difficulty locating individual items. You have already seen that at the level of individual facts or ideas, the representation of written documents has an obvious structure composed of words, sentences, and paragraphs within the series of characters, delineated through a conventional use of characters such as spaces and periods: writers and readers agree that one specific character will form a break between words, and another will form a break between sentences.

All documents (as well as other human creations) also have a more global organization, which may take almost any imaginable form. Some portions may be organized in one way and other portions in another; organization may be explicitly or implicitly marked. The important aspect is that good organization can help both the creator and the user of the document. While well-organized documents are easy to read, those created without some form of global organization can be almost useless.

6.1.1

Reading a Document

What are the really important ideas in an article? A reader who sees which ideas are truly central can better focus on those ideas. A document that leads the reader to those central ideas helps that focus. A clear structure helps a reader locate a specific section. For example, the alphabetic structure of a phone book helps readers find the needed entry; an index helps readers find topics; and the conventional location of the index helps the user find that tool.

6.1.2

Updating or Maintaining a Document

Every individual has documents that need to be kept up to date: address books, check registers, holiday greeting card lists, class

lists, “to-do” lists, etc. Sometimes the reader is the original creator of the data collection (e.g., a personal address book) and sometimes the reader merely updates a collection created by another (e.g., using the errata list that accompanies a book). Imagine the additional hassle an individual would have if these lists were simple, random collections of data.

6.1.3

Creating a Document

Although the structure of the existing objects above may be obvious, many students are surprised to realize the importance of organization on the process of creating new documents. No idea springs instantly into the mind in a complete and correct form; rather, ideas evolve. A good idea is “fleshed out” as the details become clearer. These ideas can only be expanded if the creator realizes that they are in fact the central or essential ideas. A novelist clearly must know where the novel will lead before creating the details of individual chapters. The careful data designer must first develop an organization and then provide the detailed content.

Not only do the details take longer to develop, but writers seldom develop them in the same order that a reader will want to learn them. An inventor may develop a new product and then look for an application, and a story writer may think of the conclusion before knowing how to get there. In the presentation, that writer may need to explain a problem to the reader before explaining the solution. A complicated idea requires a presentation that leads the reader through the idea in a carefully sequenced order. Writers often need to reorganize their work as they develop the material. Carefully designed structure allows such reorganization with a minimum of hassle for the designer.

6.2

Hierarchical Organizations

Perhaps the most time-honored and varied method of organization is the *hierarchical* organization. As the name implies, hierarchies are organizations with multiple levels. For example, yards are made up of feet and feet of inches; a cube has rectangular sides, each of which has edges, each of which have two endpoints; a book may have sections, each of which has chapters, which in turn are divided into sections.

6.2.1

Garden-Variety Trees

In computer science, we refer to hierarchical organizations as *trees* or *tree-structured data*. Obviously, these terms describe data or information as analogies to the biological organisms of the same name. Figure 6.1 below depicts several

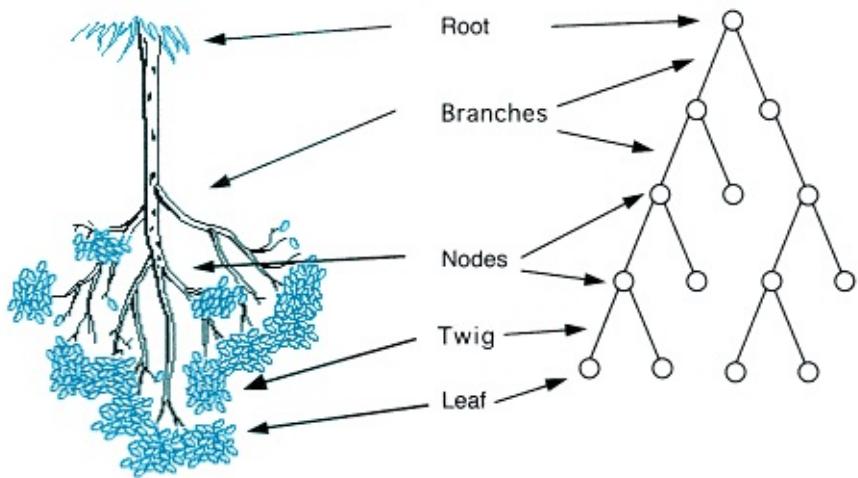


Figure 6.1
Component Parts of a Tree

additional properties of arboreal trees that we use to describe hierarchical information.

Root.

Trees are connected to the ground at a single point (perhaps trunk would have been a better term for the analogy). The *root* is the most important single part of the tree: it is the singular point from which the rest of the tree originates. Therefore, computer scientists draw trees upside down, with the all-important root at the top.

Branches.

A tree is composed of *branches*, or edges. The further one gets from the root, the more numerous are the branches.

Twigs.

The smallest (farthest from the root) branches are called *twigs*.

Nodes.

The junction of two branches is a *node*. Although this biological term is less well known than the others, it is one of the most important. A node connects a single branch toward the root with multiple branches leading away from the root. The root is also a node.

Leaves.

The nodes at the extreme ends of the twigs are the *leaves*. Generally the leaves will be at the bottom of our trees. Note that we refer to leaves as nodes even though there are no additional branches. This provides a simpler vocabulary for describing the structure. For example, every branch has a node (either a normal node or a leaf) at each end.

6.2.2

Family Tree

Perhaps the most famous analogy to the arboreal tree structure is the *family tree*, as shown in Figure 6.2. Often a family tree is even drawn exactly like a biological tree, with tree branches representing the hereditary relationships between individuals. The tree representation makes it easy to see the relationship between any two individuals. A genealogy depicts the history of one person. Each individual's parents are exactly those nodes connected to the individual by a single branch (toward the leaves). Grandparents are found in the obvious way: the parents of the parents or looking down two levels. The leaves are the most ancient ancestors. One obvious property of the genealogy tree is particularly interesting to computer scientists: each person has exactly two parents. This means that the structure of the tree is very uniform and predictable. Each level contains twice as many persons, or nodes, as the previous level. Each node has exactly two branches, mother and father. Computer scientists call this a *binary tree*. See Section 6.4 for other significant features of binary trees.

Historically, the genealogy has not been the most common form of family tree. Generally people have been more interested in a reverse tree structure: one in which the oldest known ancestor is at the root and all members of the current generation are leaves. Such a tree, sometimes called a *pedigree*, often represents

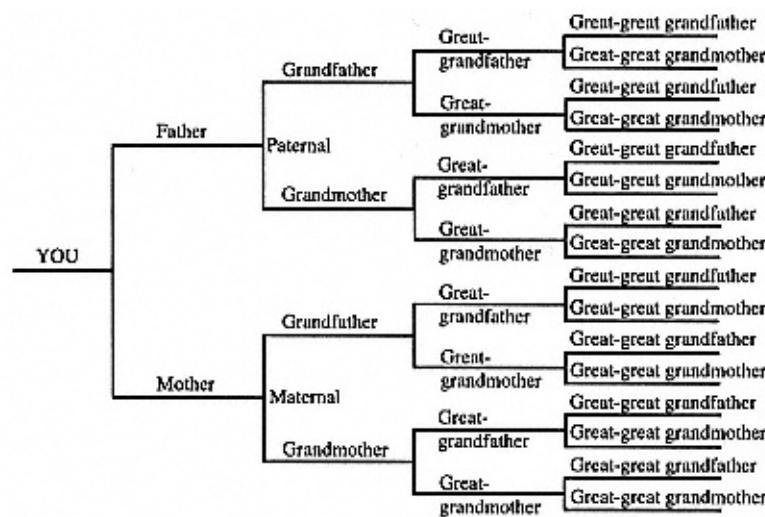


Figure 6.2
A Family Tree

only the male ancestors. Although unthinkable today, at one time many cultures felt that a pedigree contained all that was important about an individual. Pedigrees are still important to genealogists because they depict the history of the family surname, and thus the relationships among a clan.

Pedigrees are important to computer scientists for a very different reason: they provide the common nomenclature for describing the relationships between the nodes of trees as shown in Figure 6.3. If two nodes are directly connected, the

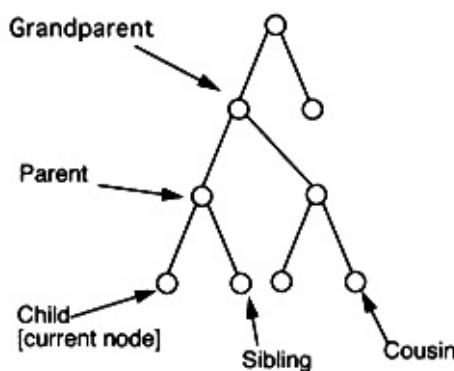


Figure 6.3
Genealogical Tree Nomenclature

parent is the node above and the *child* is the node below. *Grandparent* nodes are two levels above; *grandchildren*, two below. All children of a single parent are *siblings*.¹ If any path exists between two nodes, the upper one is called the *ancestor* and the lower one, the *descendent*. One can even extend the family relations to uncles, aunts, cousins, great-grandparents, and so on. The root is an ancestor of all other nodes. While this certainly creates a mixed metaphor with some terms based on arboreal trees and others on genealogy the collective metaphors do provide a powerful and flexible vocabulary for describing trees.

Exercises

- 6.1 Name the relationships between each pair of nodes in the tree in Figure 6.1.
- 6.2 What happens to the tree structure of a pedigree, if representations for both parents are included?
- 6.3 How many leaves are there in an ancestral tree showing three generations (the current generation at the root)? How many with four generations? Five? Six?
- 6.4 Draw your own family tree for as many generations as you know. Draw a corresponding pedigree.
- 6.5 The taxonomy used to classify living things by biologists: kingdom, phylum, class, order, family, genus, species is a hierarchical structure. Select at least three species that are not closely related. Look up their full taxonomic descriptions and draw the corresponding tree.

6.2.3

Sports Playoff

Another common hierarchy represented as a tree is the sports playoff chart such as the teams in the NCAA basketball tournament

or the Wimbledon tennis tournament. Figure 6.4 represents a playoff involving eight players. Although generally depicted horizontally, the components are logically the same as tree parts. Each node represents a contestant in a game or match. The leaves at the left represent all of the initial contestants. A parent represents the winner of a game (and therefore, a contestant in the next round). The winner of the tournament is the team at the root (right).

1. Traditionally, gender-based terms have been more common than the gender-neutral terms used here, and are still found in the literature. *Mother* and *daughter* are used instead of parent and child. Less frequently one finds *father* and *son*. Sometimes, *sister* or *brother* are used rather than sibling. Aunt or uncle is still used to describe that relationship.

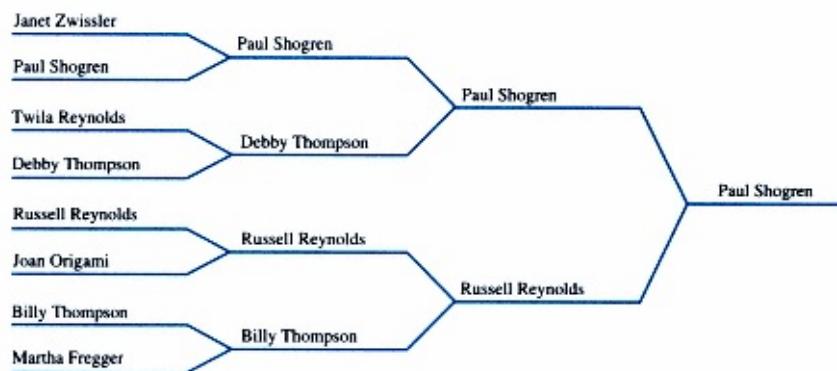


Figure 6.4
A Single-elimination Tournament

6.2.4 *Corporate Organization*

Corporations often represent their internal organization in terms of a corporate hierarchy (Figure 6.5), with the president or CEO at the top (root) and the rank-and-file employees at the bottom (leaves). Each branch represents a supervisory relationship: each supervisor is a parent and each subordinate is a child. Such a chart contains much more information than do the words and names alone. The positioning of a person or title tells as much as or more than the words themselves. Roughly, the closer a person is to the top of the tree, the higher she will be in the

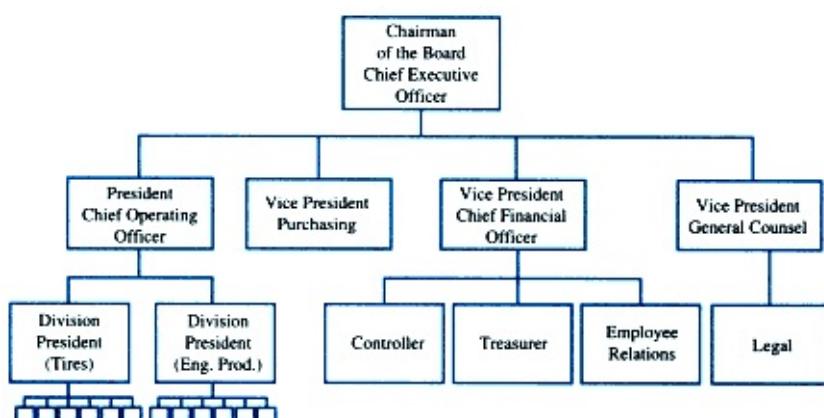


Figure 6.5
Corporate Hierarchy

chart (and the "corporate ladder"²). The tree shows exactly how many promotions are needed to get to the top, and how many other persons are equally situated. The chart also points out that the CEO is a single person (not a committee), and tells how many vice presidents the company thinks are useful and how many levels of management exist. It shows which persons are approximately equal in the organization even though they have unrelated job titles and work in different divisions.

6.2.5

Computer Folders

You have recently added yet another tree structure to your experience: directories or file hierarchies in a computer. Although directories are not usually explicitly depicted as trees, they are hierarchical structures nonetheless. Figure 6.6 depicts one user's files as a hierarchy. A directory has a single root, the desktop or disk. Below that are each of the individual disks (nodes). Each disk may contain multiple folders (also nodes); each folder can itself contain more folders (still more nodes); but some folders must ultimately contain only documents (leaves). An arrow connecting two nodes means that the child is contained in the parent. Part b of Figure 6.6 shows the more common representation of the same hierarchical structure. Notice how the folder hierarchy reflects the hierarchical relationships between the storage containers in an office: a paper is in a folder, in a drawer, in a filing cabinet, and so on (this is, of course, an important property of analogies: they should reflect the important properties).

As with the corporate ladder, only single paths through the data hierarchy may be relevant. A document is in a folder, which is in another folder, . . . , on a specific disk. This provides a means of finding or identifying a specific document. If folders are well organized, the user need not search an entire disk for a given document. If Billy wanted to recall what he was asked to do in

homework Assignment 3, he could start at the root, his own disk, then successively in the folders:

Homework assignments

Hw #3

[and finally in the document]

HW#4 handout

Section 6.2.6 generalizes this ability of trees to aid in a search.

2. Another mixed metaphor: a corporate hierarchy is tree structured, but the term “corporate ladder” describes that structure from the perspective of a single individual. From that person’s view, it is indeed a ladder. The important information is how close you are to the top. “Lateral” transfers to another job at the same rank do not get you closer to the top and would therefore not be promotions.

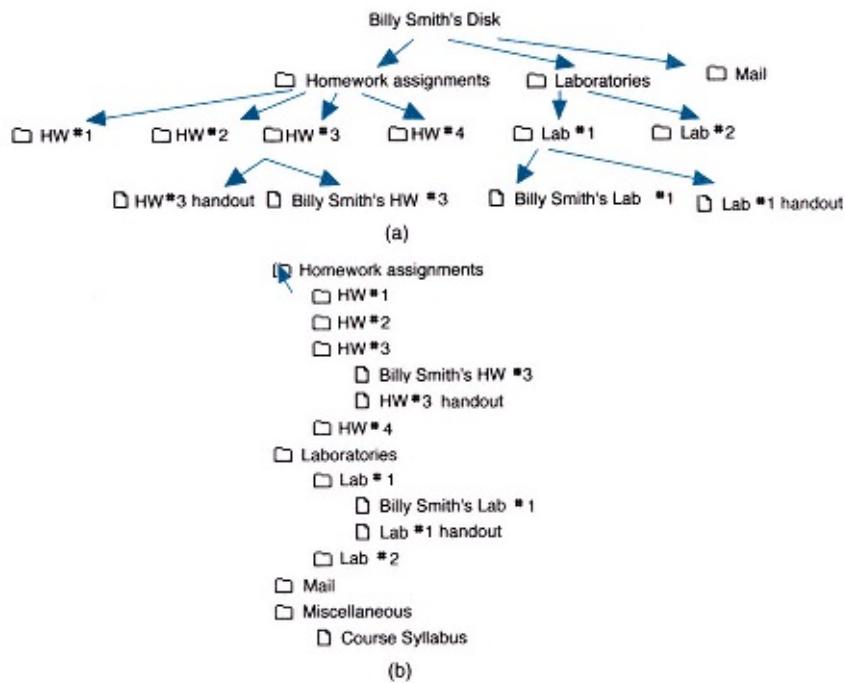


Figure 6.6
 (a) A Hierarchy of Files.
 (b) A More Common View of Billy Smith's Disk

6.2.6

Decision Trees

An important problem-solving methodology uses a *decision tree*. The methodology systematically eliminates incorrect answers by asking a series of subquestions. The answer to each of these subquestions reduces the number of possible solutions:

Repeat the following steps, until you find the answer:

Ask a subquestion.

Find the answer.

Eliminate all possibilities that do not conform to this answer.

For example, the *Pacific Coast Tree Finder* (by Tom Watts) is a field guide to trees (the arboreal kind). The finder helps the reader identify a tree by asking a series of questions:

Does it have leaves or needles?

Are the leaves pinnate or palmate?

Are they smooth or serrated?

Is the bark furrowed or smooth?

The answer to each question sends the reader to another page or section of the book. For example, it might direct the reader to page 10 if the tree has needles but to page 60 if it has leaves. At each stage the process reduces the candidate set of trees until only one tree fitting the description remains. Notice that this methodology can be described in terms of the computer science variety of tree:

Start at the root (all trees).

Answer a question with two possible answers.

If you get the first answer

then follow the left branch to a child (group of tree types)

otherwise follow the right branch to a child (group of tree types).

If the current node is a leaf (single tree type)

then the search is complete (and there are no further questions to ask);

otherwise repeat the process starting from the current node.

Section 6.4.2 expands this concept.

Exercises

6.6Sports teams from your college or university probably participate in tournaments. Find a tree representing one such tournament and identify all parts of the tree using computer science terminology (at many schools, the sports department publishes the needed information).

6.7Describe your college or university faculty in terms of a tree structure (most of the information can be found in the catalog).

6.8Build a decision tree for recognizing geometric shapes. It should include at least square, rectangle, rhombus, trapezoid, simple triangle, isosceles triangle, equilateral triangle, right triangle, pentagon, circle, and ellipse.

6.9Draw the file structure of your own floppy disk (or directory)

that you use with this course as a tree.

6.3 Outlines

But in most cases planning must be a deliberate prelude to writing. The first principle of composition, therefore, is to foresee or determine the shape of what is to come and pursue that shape.

WILLIAM STRUNK and E.B. WHITE³

My dictionary provides several definitions of *outline*, including:

1. A line marking the outer contour or boundaries of an object or figure (see Figure 6.7),
3. From Strunk and White's book *The Elements of Style*. This little guide may be the most concise and widely cited style guide for writers ever written.

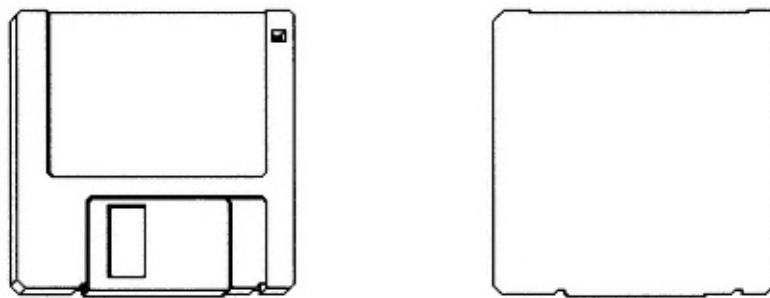


Figure 6.7
An Object and Its Outline

2. A summary of a written work or speech, usually analyzed in headings and subheadings, and

3. A preliminary draft or plan.

Actually, *Definition 2* is just an extension by analogy from *Definition 1*: An outline shows the shape of an object by providing just enough detail to see the major characteristics; a written outline lists the major characteristics. *Definition 3* suggests that an outline can be a tool for creating written work.

6.3.1

The Outline as a Tool for the Reader

The outline of a book (or even a term paper) is also a tree structure. A table of contents is actually an outline illustrating the hierarchical organization of the work (see Figure 6.8 below). The title of the book represents the root. The chapters,

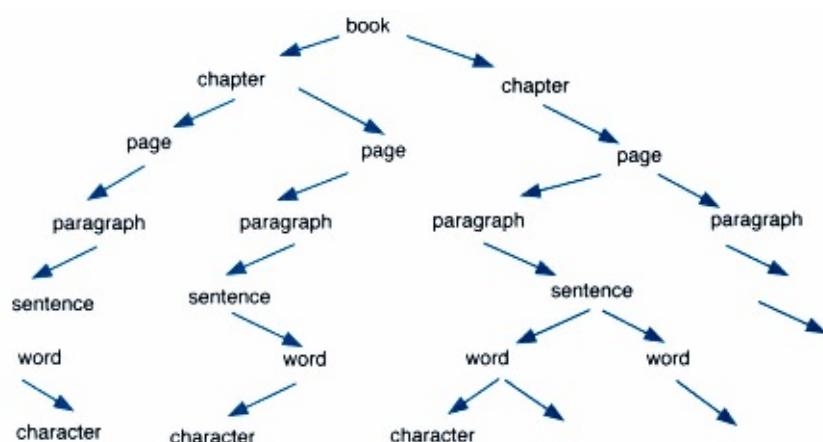


Figure 6.8
A Table of Contents as a Tree Structure

major ideas, sections, or topics represent the first level of nodes; the subsections are the next. Finally the individual sentences of the paper (which do not appear in the outline) can be thought of as the leaves.

Headers.

Headers provide the reader with an expectation of what is to come. The reader need not ask, “What is the point here?” The section header answers that question or at least provides strong hints. A reader who wishes to return to a previously covered point can do that more easily if headers distinguish the sections. This text has four levels of organization below the root (book title: *Problem Solving with Computers*); chapters (e.g., this is Chapter 6); and three levels of header (this paragraph is part of Section 6.3 and of subsection 6.3.1; the third level has no numbering, but is indicated by bold letters). This explicit representation of the hierarchical structure of the text helps the reader understand which points are major, which are minor, when the subject changes, and so on. Notice that the use of headers is simply a systematic application of the *ad hoc* techniques of the previous chapter.

The Table of Contents as an Outline.

A table of contents is really an outline of the sections, showing the structure and relationships among them. It shows the reader the overall structure of the entire document, very much as did *Definition 1* of *outline* above. Any large work will have a table of contents; users could not easily scan a document without one. One trivial implication follows from this observation: any book or other large work will require an outline eventually—you might as well do it first. Finally, notice that the table of contents is actually a collection of pointers (see footnote 1 of Chapter 0): each entry in the table of contents points to a section of the text.

6.3.2

The Outline as a Tool for Creating a Document

An outline also helps the author make sure that a paper is well structured, that it contains all of the author's important ideas, and that the ideas are presented in an order that allows the reader to follow the flow of the argument.

Completeness.

Few (if any) authors can remember all the topics they wish to present in a paper (see Box 3.1). How often have you said, "I was going to do *x*, but I forgot." This sort of error can be catastrophic in a term paper. By constructing the entire outline first, you will avoid problems caused by forgetting the individual pieces that make up the overall argument.

Ideally, a writer starts by listing the most significant ideas, and then fills in the details—the subconcepts that make up the major ideas—for each of those major ideas. Finally, for each of those sub-ideas, she finds the sub-sub-ideas—the points that make the sub-idea clear. There are several names for the approach of "fleshing in the details" by filling in the body or sub-ideas of a general outline.

Computer scientists call it the *top-down approach*: the designer starts with the most major (top) idea in the outline and works down toward lesser (lower) ideas.⁴

Logic and Structure.

Almost certainly, the initial design period is one of frequent changes of direction and correction. Consider how much easier it is to cut and paste a header than it is an entire concept. An author who concludes that one idea should appear sooner and another later, can move the items easily, by cutting and pasting at the new location. Careful use of an outline during the design process saves the writer much time that would otherwise go to correcting mistakes.

Consider an example: you are writing a term paper on Ben Franklin. After conducting your research, you know a lot about him, so you just start writing. Unfortunately, the kite-and-thunderstorm incident slips your mind. You then have to go back and rewrite the paper including the incident. “Ah ha,” you say, “That’s what word processors are for.” All the writer needs to do is go back and insert the missing sentences.” Unfortunately, modifications of document structure force corresponding modifications of sentence structure.

Pronouns cannot be used until they have an antecedent.

Back references depend on the order of introduction.

Logical structure and flow of events may be independent.

Repeated words in consecutive sentences causes monotony.

Description of events may change tense, number, or even person.

The threads that hold events together can get lost.

Unfortunately no word processor can help much with such changes. But organizing the paper in advance can. The paper should have some basic top-level organization or theme, perhaps:

Franklin the inventor,
Franklin the scientist,
Franklin the statesman, or
Franklin the ladies' man.

Each may have exactly the same events, but the structure will vary. If the designer changes her mind while the paper is still an outline, she only needs to move the topics. If she changes it after writing the text, she needs to deal with all of the problems above. The bottom line: always create an outline before writing the document.

Styles of Outlines.

Any structured approach to a written document may be an outline. Unlike the *ad hoc* methods of Section 5.3.1, an outline has certain standard representations such as: Harvard (successive levels indicated by Roman numerals, upper case letters, arabic numerals, lower case letters, and so on) or

4. The concept of top-down design recurs throughout the remainder of this book. Watch for it, particularly in Chapters 11 and 24.

legal (the numbering in this text). Some environments require specific styles. In almost all cases, the style should be carried through an entire document. Most outline tools provide support for styles, automatically maintaining the style throughout the document.

6.3.3

Outlines and Computerized Documents

Paper outlines work reasonably well. However, the outline almost always becomes a convoluted network of insertions, deletions, and arrows indicating where each item should now be. It will certainly be necessary to rewrite a paper outline more than once. Consider the following problem: An outline initially has the structure:

- I.
- II.
- A.
- B.
- III.

but an improved version groups items II and III together, thus making the structure

- I.
- II. (new node)
 - A. (old II.)
 - 1. (old II.A.)
 - 2. (old II.B.)
 - B. (old III.)

What are the individual changes? Both the letter designations and the indentation changed for almost every item in the list. Fortunately the new numbering and organization are quite predictable. It is just a matter of human time to make all of the changes: inserting new tabs and rewriting the labels. But predictable changes are what computers excel at.

In conjunction with a computerized word processor, outlines

become even more useful. *Outliners* are word processing tools with added features designed specifically for working with the outline structure of a document:

node-level cut and paste facilities

insertion of new items (nodes) or sub-items

automatic restructure of document when a node is added, deleted, or moved.

Some outliners are independent applications; others are parts of larger word processing systems. But the essential features are the same. They have a higher granularity: the natural unit for manipulating (constructing or modifying) an outline is the node (line). The most common actions are moving an entire node, and perhaps the entire subtree (section) headed by a node. In a word processor, the basic unit is the word. It is more difficult to select an English sentence. Outliners,

in contrast, provide tools to select, cut, copy, or move an entire node. In fact, this is usually easier than copying a group of characters in a text processor.

On the other hand, outliner operations are considerably finer grained than those of operating systems or mail tools. They work best on a section of the document: select a single node, move that node to another place, reorder the document, and change the level of a node.

Basic Outliner Operations.

The basic operations dramatically increase the text manipulating power of a word processor. In a basic word processor, the task of moving text from one location to another is a four-step operation:

*Select the text to be moved,
Cut the text from its original location,
Select the new location, and
Paste the text into that new location.*

In an outliner, this is reduced to two steps

*Select the entry, and
Place it at its new location*

Typically, this is accomplished by grabbing the entry and dragging it to the new location. No explicit cut and paste are needed; the object is moved directly to its new location. Selecting any part of an item selects the entire item. There is no need to specify the beginning and end of the entry. The outliner assumes that most operations will be performed on entire entries.

The Trade-Off.

Of course, any improvement has a price. In the case of the outliner the price is that operations on objects of small granularity (e.g., characters) are more difficult. Outliners are usually more awkward

for accomplishing basic text operations, such as changing the font or the style of a word. The conclusion: at any point in a task, use the tool most appropriate for the immediate task at hand. In the case of writing, this frequently means:

Develop and modify the outline using an outline tool.

Fill in the body using a word processing tool.

Exercises

6.10 One common means for selecting a larger object in an outline tool is a triple click. Try it on your outline tool to see if it works. Also try it on your word processor and note the similarity or difference.

6.11 In most outline tools, the sequential ordering can be changed by dragging a node up or down the outline listing. The level of a node can be changed by moving the node horizontally to align with other nodes of similar position. Try this in your outline tool.

6.4

Trees and Computer Science

Trees are one of the most important structures in computer science. Tree-structured data seems to be omnipresent. The following subsections discuss some of the important characteristics of trees from the computer science perspective.

6.4.1

The Definition of Tree

Trees can be defined using an incredibly simple technique, *recursion*, a technique that appears throughout the discipline. Consider the tree in Figure 6.1. What would be left if you threw away the root? Two sub-trees (numbered 1 and 2), smaller trees, but trees nonetheless. The basic characteristics still hold: each tree has a root and children. The process of removing the root to create two smaller trees could be repeated until some root has no children. Would such a root be a tree? Yes. This is an example (like the null string) in which the use of degenerate examples makes the definition simpler.

Now, reverse the process. Given any two trees, you can define a single larger tree by creating a new root and connecting that root to the roots of each of the original trees. If degenerate trees are allowed, this also describes how to build the simplest tree that looks like a tree: Given two individual nodes (by definition, each is also a leaf or a tree), create a new root with each of these two single-node trees as children. Any tree can be defined in this way. Therefore, computer scientists use these notions as the basis of their definition of tree:

A structure is a tree if it is either

- (a) *a single node, or*
- (b) *a single node with one or more child nodes (each of which is the root of a tree).*

Thus the root in Figure 6.9 is a tree by part (a), as are new roots 1 and 2.

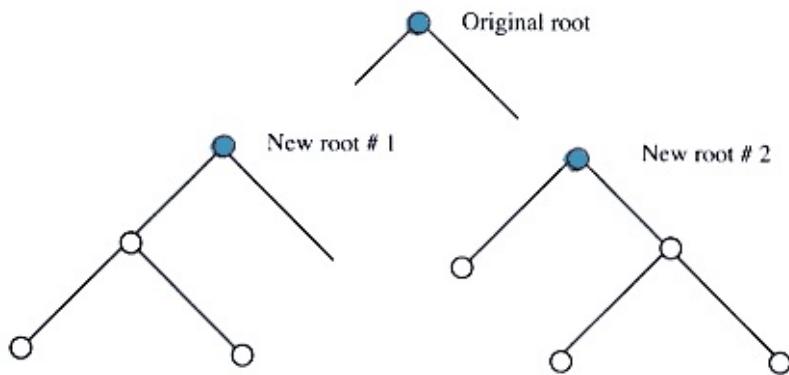


Figure 6.9
Component Parts of a Tree

The leaves at the bottom are trees by part (b). (Such definitions appear again in Section 16.1.)

6.4.2

Trees and Searching

The decision trees of Section 6.2.6 are also common tools occurring in many computer science situations, such as networks of computers and representations of computer programs. From the computer scientist's point of view, one of the important properties of trees is the creation of level. In most trees, the number of levels is much smaller than the total number of nodes. When computer scientists say "much smaller," they actually mean "very much smaller." For example, a binary tree with a million nodes needs only twenty levels from root to leaf. (For those of you who are mathematically inclined, a tree n levels high can have 2^n nodes: in this example, 2^{20} is approximately one million). This makes trees an important tool for finding one specific item out of a large group of similar items. Suppose you wanted to play the children's game of "Guess what number I am thinking of," in which one person says, "I am thinking of a number between one and a hundred. Guess what it is." The second person must then guess numbers, while the first one answers "too high," "too low," or "that's it!" A player making random guesses would take, on the average, fifty tries to guess the correct number. A more intelligent approach, called a *binary search*, models the possible answers as a binary tree structure. The player guesses a number in the middle, and treats that as the root. Then, depending on the answer to the guess, the player guesses the root of one subtree or the other. Figure 6.10 shows the series of attempts in which the correct answer is 15 and the guesser gets there in five guesses. This strategy guarantees a correct answer in no more than seven guesses ($2^7 = 128$, which is obviously greater than 100). Although few computer programs play the guessing game, exactly the same technique is useful for many real world problems. Any problem of locating one item out of many is a

candidate for this

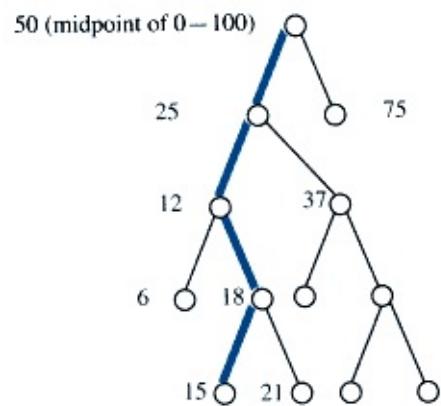


Figure 6.10
Finding the Number 15

approach. It is certainly the best technique for finding a given entry in the phone book: start by looking near the middle and then moving forward or backward depending on whether the name you want is alphabetically before or after your current location. In the computer world, binary searches may be used by the system for tasks such as finding a named document on a disk.

Exercises

- 6.12 A degenerate tree, in which each node has only one child, would have as many levels as nodes. Most people would even hesitate to call it a tree, but it does fit the definition. Draw such a tree. Label all parts. How many leaves does it have?
- 6.13 Build an outline of the computer applications you have seen thus far, using their granularity as a definition for intermediate nodes.
- 6.14 How many nodes are there in a binary tree with 2 levels (including the root)? How many nodes with 3 levels? 4? 5? 6? 10?
- 6.15 Another way of stating the mathematical relationship of Section 6.4.2 is that the logarithm of $128 = 7$. These are base 2 logarithms and you have probably used base 10 or base e logs. Many students are taught logs but never see any reason for using them. This provides one reason: to describe the height of a tree containing n nodes it requires $\log(n)$ levels. (These ideas reappear in *Chapter 15: Iteration: replicated structures.*) How many steps should it take to guess a number between 1 and 1000?
- 6.16 What is the number of rounds needed to produce a winner in a 64-team tournament?
- 6.17 Describe the process of looking a name up in the phone book as a decision tree.

6.18 Count (or estimate) the number of entries on a page of your phone book. Use the techniques of Exercises 6.15 and 6.17 to determine the number of guesses you might need to find the entry for Juan Garcia (assuming you were on the right page).

6.19 If the telephone book has 500,000 entries, how many steps would it take to find a specific entry? What would it take using brute force?

6.5

Structures Other Than Hierarchical

Trees may be one of the most important data structures in computer science, but they are not the only ones. You will see at least three more in this course.

6.5.1

Lists

Sequences of objects are the simplest of the common structures. Text, viewed as a stream of characters, is a *list*, and so is a sequence of numbers to be added up.

6.5.2

Tables

Not all information is hierarchical. For example, *tables* are two-dimensional structures resembling chessboards. They can be used for storing anything from the multiplication tables to compatibility charts showing possible combinations of products. *Chapter 8: Relations, Tables, and Databases* begins an extensive study of both lists and table-structured information.

6.5.3

Graphs, Pointers, and Multiple Organizations

A *graph* resembles a tree but is defined with less rigor. For example, a child can have two parents, or even be its own parent. Electronic media offer a structure not really possible in traditional media: *hypermedia*. In hypermedia, multiple structures are superimposed on a single data set. To a limited extent, the *Gateway Labs* accompanying this text have multiple organizations. To a lesser extent, so does a well-indexed book. In all cases, the pointer serves as the essential glue for creating the structure. *Chapter 21: Graphs and Hypermedia* discusses multiply organized data.

6.6

Summary

Hierarchical or tree structures are incredibly common. For example, both computer file systems and most written documents have a tree structure. Understanding this structure helps reduce *thrashing*⁵ in your work. A written outline is simply one instance of the concept of tree. Outliners are tools that help writers take advantage of the tree structure to organize their work.

Important Ideas

tree sibling

branch
binary
search

child node

outline parent

root hierarchy

leaf

5. *Thrashing* is a technical term in computer science, usually used in reference to disk storage. A computer with a very full disk can reach the point where it literally does nothing but move data around on the disk: it attempts to move *A* to make room for *B*, but then needs to move *B* to make room for *A*, and. . . . The same principle applies to humans who spend so much time rewriting a document that they never complete the document.

Ideas that will Reappear

granularitytable

pointers list

recursion top-down
design

graph

7

Models, Visualization, and Pictures

I would give all the wealth of the world, and all the deeds of all the heroes, for one true vision.

HENRY DAVID THOREAU

Pointers

Gateway *Paint tools*

Labs: *Draw tools*

Lab Unit 6: Models and

Manual: Visualization

7.0

Overview

A picture is worth a thousand words.

Or, at least that's what they say. The current question is, "Is it really?" This chapter explores the use of pictures as tools for transmitting information or for improving your own understanding. Creating good illustrations requires an understanding of both the concept to be illustrated and the techniques for illustrating. In particular, selection of an appropriate model of objects and images is an essential first step for creating illustrations.

You can't see the forest for the trees!

Think about this familiar expression for a moment. Restate it in a less folksy, but more specific, manner:

You have focused your attention so much on the little details that you cannot see the big picture.

or perhaps:

You know all of the specifics but cannot create a generalization for the whole.

Tools that help people see generalizations or “the big picture” are collectively called *visualization* tools. Obviously this must include tools for drawing pictures,

but it also includes the concept of modeling itself (see Section 0.2.1, which describes modeling as one of the three central themes of computer science). That is, selection of a model is an essential aspect of visualization. Chapter 3 shows that one's understanding of a concept depends on the particular model employed. This chapter illustrates the central role of models by describing the impact of the visual model on the use of tools for creating visual models.

7.1

The Advantage of Pictures

The invention of the written alphabet and of writing itself are considered to be among the greatest human achievements. Yet written and spoken language has its limits. Some concepts are not communicated nearly so well by text as they are by other means. When given the task:

Describe a goatee

most people instinctively bring their hand to their chin to help illustrate the beard's location. (Try this on your friends.) Apparently most people find that the hand is much better for locating than words are.¹ Pictures have several properties that make them better than words for communicating some forms of information.

Analog.

Pictures provide *analog* models. That is, the distance between two points in a picture reflects the distance between two real world items. Points that are further apart represent objects that are further apart. The hands of a traditional clock move in an analog fashion: the further they move, the more time has elapsed. A person who needs to know that it is approaching four o'clock will probably find the clock in Figure 7.1 (a) more useful than the one in Figure 7.1 (b). The distance between any two points on a map of a city is proportional to the distance between the corresponding locations in the city itself. Distance is a



(a) analog clock



(b) digital clock

Figure 7.1
A Picture of Time

1. Actually, you have seen this concept already: a mouse is better for describing locationsselecting positions of objects or regionsthan words (commands) are.

continuum: two objects can be any of an infinite number of distances apart. A picture can reflect that possibility. Written language has no nice way of expressing small variations in distance.

Dimensionality.

Pictures are inherently two-dimensional. And well-known techniques exist for representing three dimensions in a picture. Written language is essentially one-dimensional. Suppose you wanted to list (or even count) the states bordering Texas. The task is easy given the map in Figure 7.2, but would be very difficult given only a verbal description of the fifty states.

Identification.

Because a picture places information directly in front of the reader, it is easier to indicate specific details. Consider how easy it would be to indicate the location of South Dakota in the figure with the addition of a single arrow.

Color.

What color is mauve, magenta, or cyan? There are far more colors than we have words for. The dimensionality of pictures is easily extended by the addition of color.

Exercises

7.1 Work with a friend. For each of the distances: inch, foot, centimeter, decimeter, and cubit, independently draw a line of that length without using a measuring device. Compare the two sets of lines. How much variation was there? (Note that the important result here is not how far you are from some absolute measure, but how far you are from each other.)

(table continued on next page)



Figure 7.2
A Map is Two-Dimensional

(table continued from previous page)

7.2 Compare visual images and smells. Which are more complex? Which can be described in words more easily? Why?

7.3 List five measuring tools. Describe each as analog or nonanalog.

7.2

Objects and Images

The world is composed of objects: cars, computers, boxes, other people, and so on. People see those objects as what they are. But the images of the objects are quite distinct from the objects themselves. The retina of the human eye contains millions of cones, each of which receives the image of a single point² in the total picture. The human brain translates this collection of individual points into much larger lines and objects by associating individual points. Newspaper pictures and television images take advantage of this translation. Each is a collection of individual points called *pixels*. The human eye translates the printed pixels into objects in almost exactly the same way as it would the individual point-images coming from the cones in the eye.

An image on a computer screen, like a newspaper picture or a television screen, is composed of many pixels. A computer creates an image by coloring many individual pixels. For example, an arrow drawn on the screen may appear as in Figure 7.3 (a). But it is actually a collection of individually colored pixels, as in the magnified view in Figure 7.3 (b). Notice how the apparently straight edges of the arrow head are actually jagged, following the edges of the individual pixels. This “staircase” result of forcing the actual points of the image to align with the discrete rows and columns of pixels is called *aliasing*. The smaller and closer

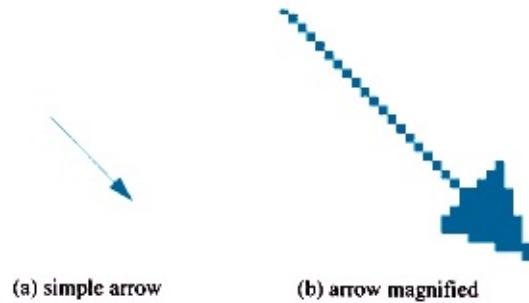


Figure 7.3
Two Views of an Arrow

2. I use point here as “a very small two-dimensional image,” not as the no-dimensional concept of mathematics.

together the pixels, the less obvious is the aliasing to the human eye. Newspaper or television pictures have exactly the same property (look at a light gray area for best contrast, or use a magnifying glass). You can even see the individual pixels on a computer screen. This is especially easy on older machines because the relatively few pixels were relatively large. On newer computers, the individual pixels are smaller and harder to see. Try looking at text characters (italicized words are often especially good for this purpose), curved lines, or almost horizontal lines. The pixels of a computer screen are arranged in rows (for example, the small *Mac Classic* screen had 352 horizontal rows of 512 pixels each; larger screens may have 1000 or more rows, each with 1000 or more pixels).

7.3

The Models Define the Tools

Computerized visualization tools come in at least two formsone based on each of the two models of images: entire objects themselves and the pixel-representation of those objects. The best tool to use will depend on factors ranging from availability to the nature of the drawing you wish to construct. Oddly enough, each model is easier to understand if you also understand the other.

On a quick inspection, the tools are almost identical. Each is a collection of illustration tools, similar to the draftsman's T-square, triangle, and compass. With either class of tool, users create individual objects: rectangles, ovals, lines, and so on. In each, the user uses a collection of tools (called a *palette*, as in Figure 7.4) for creating the individual images: lines, rectangles, arcs, polygons, and so on. Each type of application can draw a similar collection of objects. Each can draw objects as simple outlines or fill them in with colors or patterns.

The two types of visualization tools interpret the objects they create differently, however. One class, *draw tools*, views an image as an

object similar to objects the human brain perceives (e.g., lines and squares). *Paint tools* interpret the

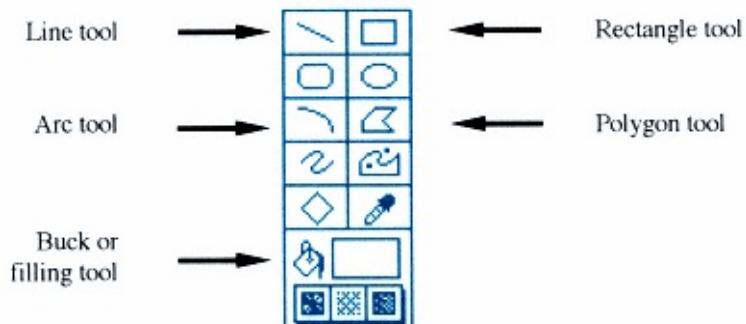


Figure 7.4
A Typical Tool Palette

created image as a collection of pixels like the computer screen, the newspaper picture, or the human retina. This difference in interpretation directly impacts the ways in which the image can be manipulated.

7.3.1

Paint Tools

The first class of tool, called *paint tools*,³ models an image as a collection of pixels. Paint tools treat an image as just so many individual pixels, just like a drawing created with pencil and paper. Consider a line created using a straightedge and paper. The human eye can see that it is a line, but the artist cannot further manipulate (e.g., move or stretch) the line as a single object. She can, however, alter the appearance of the line by erasing part of it or adding additional lines. Because the image is a collection of pixels, the artist has complete freedom to draw dashed lines, erase part of a line, draw a section freehand, and add shading and other individual touches. Any variation is possible by adding or deleting individual pixels. Compare paint tools to “paint-by-numbers” sets. The unpainted canvas looks like a jumble of labeled regions, but collectively the colored regions form a coherent image.

Using the pixel-oriented paint model, an architect might create the drawing of the room in Figure 7.5 by:

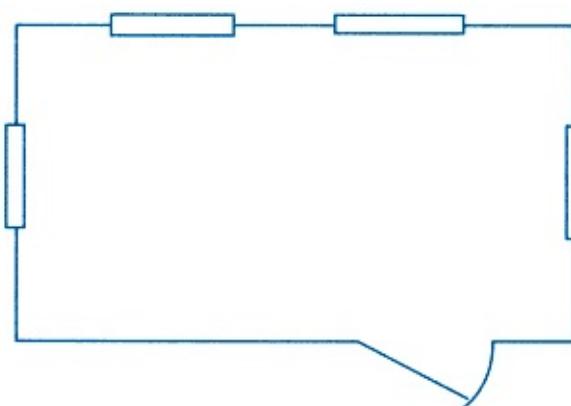


Figure 7.5
A Simple View of a Room

3. As with word processors, the general principles of paint tools apply to all paint tools, including *MacPaint*, *SuperPaint*, *DeskPaint*, *MS Windows Paintbrush* and the paint tools contained in integrated packages such as *ClarisWorks* or *HyperCard*. Unfortunately, observations based on paint applications may not be applicable to draw applications. Notice that the product name often indicates which model it is based on.

First draw the room as a rectangle.

For each window:

Erase the part of the rectangle where the window goes.

Draw a simple rectangle representing a window.

For the door:

Erase the part of the rectangle where the door goes.

Add the individual parts of the door:

To make the arc

draw a circle and

then erase the parts not needed.

Draw a line from the wall to the arc.

Because the picture is composed of pixels, erasing sections of the rectangle is easy: erase the individual pixels as needed. Most paint tools even allow artists to magnify or *zoom in* on the picture, making it easier to add and delete specific pixels.

On the other hand, the tool provides no identification of pixels with an individual object. The fact that the artist colored a group of pixels as a line, rectangle, oval, or other well-defined shape is not relevant. Since the image is just a collection of pixels, there is no easy way in Figure 7.5 to move the entire window or door to a new location on the wall, or change its size. Adding or changing an individual line may create very sloppy appearances, as in the right-hand images of Figure 7.6. The upper line was not extended straight; an attempt to make the lower one thicker resulted in a line with one end wider than the other.

Paint Operations.

Typical paint tools allow the user to paint a shape (rectangle, oval, line), color in a bounded region, erase pixels, and copy or move a region.



Figure 7.6
Problems Extending a Line in a "Paint System"

Exercises

7.4 Practice with your paint tool. Create one of each basic shape provided in the tool palette. Erase one shape. Does the tool support color? Can you fill in areas? (Hint: remember to use the Help facility.)

7.5 Use a paint tool to draw a picture illustrating your hobby.

7.3.2

Draw Tools

With a *draw tool*,⁴ the artist creates shapes, or *objects*, in exactly the same manner as with a paint tool. However, the objects themselves are the basis of the model. The shapes, not the pixels, are the primary building blocks used to construct the full picture. Each object retains its integrity and therefore can itself be manipulated as an entity. Think of the objects as paper cut-outs if you like. Consider the two rectangles in Figure 7.7. Since the two objects maintain their identity, the user can manipulate them individually. For example, the user can transform Figure 7.7 (a) into (b) by moving the striped rectangle from the front to the back. In general an object may be moved to a new location, stretched, rotated, inverted, or filled with color. For example, the user can create a square⁵ with the “rectangle tool” (looks like a rectangle on the palette) and then transform it into any of the rectangles in Figure 7.8. The manipulation preserves the integrity of a rectangle. You can make it bigger, smaller, fatter, or narrower. You can also make the edges thicker or thinner, color it, fill it in with a pattern, or even rotate it. On the other hand, all objects must retain their identity. You cannot change a rectangle into a triangle or a



(a) Striped rectangle in front



(b) Striped rectangle in back

Figure 7.7
Distinct Objects in a “Draw” System

4. As with word processors and paint tools, the general principles of draw tools apply to all draw tools, including *MacDraw*, *Canvas*, *DeskDraw*, and the draw tools contained in integrated packages such as *ClarisWorks* or *Microsoft Works*. Again, the name often provides a hint about the model used by a product.
5. Recall that a square is, after all, just a rectangle with adjacent edges equal.

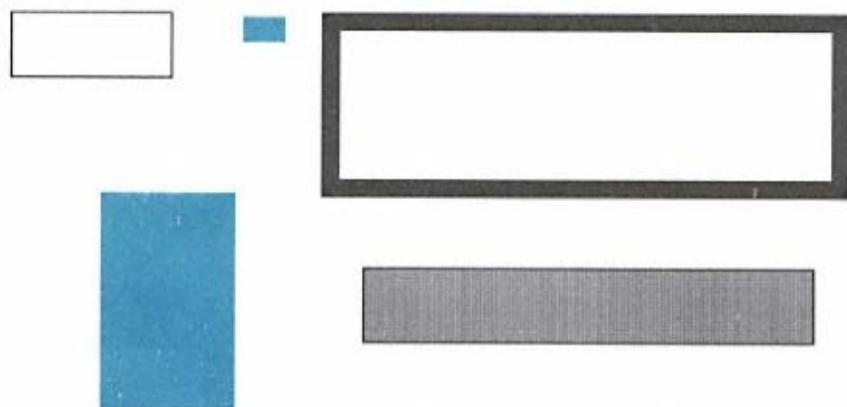


Figure 7.8
Manipulating an Object

circle, erase half of it, or break it into two parts. You cannot erase an erratic path through a rectangle as in Figure 7.9 (erasing the swath was easy with a paint tool: just use the `Erase` tool to erase the individual pixels). Similarly you cannot create a semicircle by erasing half of a circle in a draw tool. As a result, the architectural drawing of Figure 7.5 creates problems for a draw tool: there is no nice way to make the doorway by erasing part of the rectangle.

Exercises

7.6 Repeat Exercise 7.4 and Exercise 7.5, but use a draw tool.

7.3.3

Comparing the Models

Granularity Revisited.

Just as the various forms of text processing tools are distinguished largely by differing levels of granularity (e.g., character level for

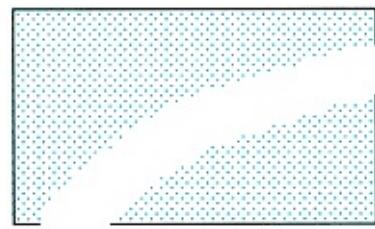


Figure 7.9
A Difficult Operation in
a “Draw” System

word processors, lines for outliners, entire documents for mail systems), draw and paint systems are also distinguished by this measure. The granularity of the paint system is the pixel and the granularity of the draw system is the object. Thus paint systems tend to be more useful in other situations requiring fine resolution, such as photographic or artistic rendering. Draw systems may be more useful for situations involving variations on complex objects.

Manipulating Regions.

In a draw tool the user creates a complex drawing by associating objects together. This makes it relatively easy to manipulate a complex object, such as moving a person from one part of a picture to another. In a paint tool the user selects all of the pixels within a *region* (usually a rectangular area or outlined shape). This makes it easy to manipulate an area of a picture, for example, to color the petals of a flower.

Bit-Level Operations.

Because paint tools manipulate pixels directly, they typically have several sub-tools which have no direct equivalent in a draw tool. In particular they can erase an individual pixel or add pixels individually. For example it may have a spray paint tool that randomly places bits in a region. Just like spraying paint on a house, the longer the user spray paints, the denser and more evenly painted the region becomes.

Hidden Objects.

A paint tool does not really support the concept of *in front of*. When it places one object in front of another, the pixels comprising that object replace the pixels of the original object.

Specifying an Operation.

Recall that the full description of an operation requires both the operation itself and the object. But in a paint tool large objects lose

their identity after the object is painted. The user must therefore select the color of an object or line before painting it. In contrast, the user of a draw tool may select an entire existing object and change its color. A paint tool user can select any pixel and manipulate it, but the draw tool user will need to create an object and place it at that pixel.

Selection of an Object.

In a word processor, you select any text you wish to manipulate. Text is thought of as one-dimensional: simply drag from the beginning to the end of the block of text. Manipulate two-dimensional objects similarly. Select objects in a draw tool by clicking on the individual objects; in a paint tool, by dragging the mouse across the region. But think about the granularity. In a paint tool, selected items are pixels, in a draw tool they are entire objects. Compare the results of a select and drag combination in the two models as shown in Figure 7.10.

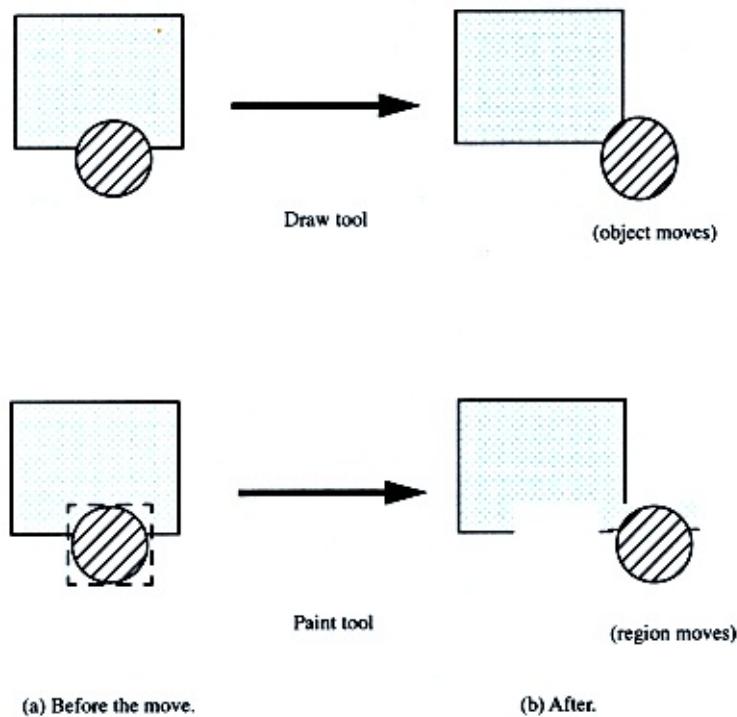


Figure 7.10
Moving Selected Objects or Regions

7.3.4

Selecting the Correct Tool for Visualization

Should you use a painting tool or a drawing tool? For many users this is not a meaningful question because they only have one tool or the other. Other users will develop a strong preference for one over the other and will always use that tool in any circumstance. But for those persons who have access to both tools and the personal flexibility and willingness to use both, the following rules of thumb may help guide their selection of a tool for a given task. In general, use the draw tool when you:

- want very geometric pictures such as lines and shapes,
- have a drawing composed of identifiable objects,
- have lots of similar objects, or
- must edit by moving or stretching objects.

Use the paint tool when you:
are creating a more art-like illustration,

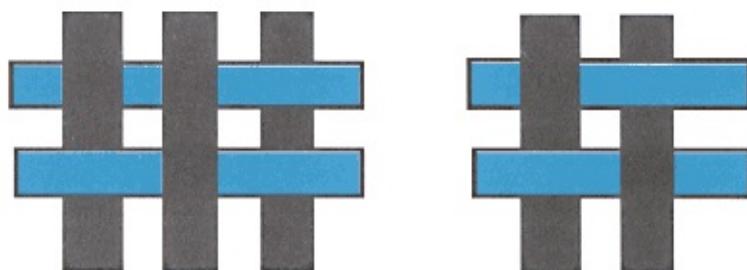
have lots of irregular or custom shapes,
need to edit the picture on a pixel by pixel level, or
want special effects such as brush strokes or spray paint.

Exercises

- 7.7(a) Use the drawing tools to create each of the basic shapes;
use the manipulation tools to modify the shapes in all the ways
you can find. Identify the other drawing tools from the palette.
Use the help tools provided.
- (b) Use each type of tool to create a drawing of a spread out
stack of cards. Then color each card a different color.



- 7.8 Use both the paint and draw tools to create each of the two basket weaves shown below. Comment on the relative usefulness of the two tools for the two shapes.



- 7.9 Section 7.3.2 said it would be difficult to create Figure 7.5 using a draw tool. But it is possible. Use your draw tool to create a similar picture. (Hint: a rectangle can be all white the same color as the page.)

7.4

Extending Text Concepts to Visualization Tools

If there is one essential piece of advice that every computer user should receive, it is:

Don't forget what you already know (from your studies of other applications).

A text processing tool manipulates only one sort of entity: characters (or groups of characters). At any given point, the user is either adding new characters or

modifying the appearance of existing characters. In an illustration system, the user may manipulate any of several kinds of objects, geometric shapes, pixels, segments of text, or groups of these things. The possible actions at any given time depend on the type of object and the tool.

Location.

In a text processor, any new text is entered at the insertion point. Normally this point is immediately after the last text previously entered. In a two-dimensional system, the concept of “after” is less well defined. Therefore, each new object must be placed somewhere as it is created. Pixels or objects can be moved to any new location.

Size.

Notice that the act of creating an object in space defines its size as well as where it will be placed.

Multiple Tools for Multiple Object Types.

In a text system all objects are characters, so a single tool (the keyboard) can be used to create any character. But a picture tool provides many shapes. A separate tool is needed for creating each class of objects. Just as modifying the object in a text processor has two steps:

Select the object.

Modify the object.

creating a visual object also has two substeps:

Select a tool.

Create the object.

Think about that for a moment. In a draw program a new object is a single new object that can be further manipulated. In a paint program, a new object is really a collection of pixels that can move

individually.

7.5

Documents with Mixed Forms

The first several sections of this chapter have a sort of schizophrenic aspect. On the one hand, they promote pictures as an improvement over or enhancement to written documents. On the other hand, they talk as if the document is composed solely of a picture. Actually, almost any document can combine both text and pictures. The document will still be essentially a text document or a picture document, but can contain objects of the other sort. Each tool provides nice facilities for working with its own type of document and not-so-nice facilities for working with other kinds of documents. That is, although a text processing application may include drawing tools, they will usually be primitive when compared with either the text processing capabilities provided by that application or the drawing tools provided by any drawing program. Whether you include pictures in a text document or text in a picture should depend on which is the most

important or largest part of the document taken as a whole. A picture of a factory that includes the name of the company written on its facade, or even a caption under the picture is essentially a picture and it should be a picture document. A term paper that includes some illustrations is primarily a text document. Selecting the correct tool guarantees that you will most often have the right features available when you need them.

7.5.1

Mixed-Form Documents

Generally you can always cut and paste. You can select and copy a picture from a picture processing application and paste it into almost any text processor document. You can almost always copy a segment of text from a word processor and paste it into a picture processing tool. In fact, as you will see as you progress through the remaining chapters, you can usually paste portions of any given document into documents of another type. Sometimes a document, or portion of a document, can be copied into another document through more sophisticated means. A document is said to be *imported* into another if the source is actually converted to the format of the target document.

7.6

When is a Picture Useful?

Pictures provide an excellent means of enhancing a document. True, there are some very trivial reasons for including pictures: such as making a term paper longer or you happen to have a pretty picture available. But there are very good reasons for using pictures, such as improving the clarity of the explanation or writing something that is hard to put into words. Sometimes the picture is quicker or more direct, sometimes it makes references easier. As general rules of thumb, pictures in documents are useful if:

they communicate information better than natural language. Any

time you find yourself waving your hands in an otherwise verbal interchange, ask yourself if you are really attempting to draw a picture.

the information is positional or geometric.

you are describing aspects of physical objects. For example the classic “Put tab *a* into slot *b*” instruction is much easier to understand if you have a picture of the objects containing the tab and the slot.

an item is classically represented as an illustration. There is probably a good reason for that. For example a map has several advantages over English instructions. It provides an abundance of detail, but the user need only take as much as is needed (e.g., most cross streets can be ignored) as well as being an analog model. It is easier to transfer attention from one point to another in a two-dimensional drawing. Finally, the natural language descriptions often leave the speaker hunting for descriptive words (e.g., “the second right turn after three oak trees on the left”).

you need to break up long blocks of text to keep the readers' attention.

you want to provide thematic indicators. For example this text uses the symbols



to indicate important items, such as pointers and exercises.

a picture can help the reader identify the topic. For example, the writer of an article on American politics might create two paragraphs, one headed by a donkey and one by an elephant. The relationship between the two paragraphs would be clear to any reader.

you want to illustrate change or transition. It is far easier to show a change than to describe it (that's one reason for the ubiquitous "before and after" pictures).

7.6.1

Times Not to Use Pictures

There are also times when you should not include pictures within a text document. These are harder to identify than the times to use them. But a few questions provide rules of thumb for the decision:

Does the picture actually add to the document in an essential way? Specifically, "What does it contribute beyond the contributions of the written text?"

Is the purpose of the picture is to be cute? Is that appropriate?

Is a picture appropriate in the context? For example, a picture may not be appropriate in a résumé or a legal document.

Can you actually make a good looking picture? Or does the resulting work look like the professional writer teamed up with an

amateur artist?

7.7

Related Visualization Tools

The importance of visualization has spawned several related products useful for illustrating documents.

7.7.1

Clip Art

Any context has recurring themes and recurring objects. Any book or paper on computing may need a picture of a computer; an article on sports may need pictures of athletes. Rather than reinventing the wheel (drawing every new picture from scratch), many users prefer to use *clip art*, a collection of pictures that can be *clipped* (cut, copied) from their source document and pasted into any other document. Clip art reduces the task of creating detailed pictures. In a very real sense, it increases the complexity of the basic building blocks from pixels or rectangles to individual premade pictures.⁶ Clip art pictures range in complexity

6. Recall an earlier comment that computer scientists are lazy and that is good. This is an example of the payoff of being lazy. Once created, objects can be reused over and over with little additional overhead.

from *dingbats* (♦*♦*) to photographic images. Building a complex image up from small pre-existing images may be likened to building a written document from existing type characters but with a much larger set of primitive characters. When using clip art be sure to check any copyright restrictions applicable to the clip art collection.

7.7.2

Image Viewers

Many specialized applications exist for viewing or manipulating complex images such as photographs or even videos. Most users use these tools as a means of viewing existing documents, but many also create complex original documents using the same tools. These viewing programs process the complex internal formats of the images so that the user need not be concerned with the fine details of representation. The list of internal formats is continually changing common formats presently include *BMP*, *JPEG*, *PICT*, *GIF*, and *TIFF*.⁸

7.7.3

Scanners

A scanner is a tool for copying an existing image from an external source, such as a magazine picture, photograph, or a child's crayon drawing, into a computer document. A scanner creates an entire computer image in a single step rather than drawing a new one from scratch. They work on almost exactly the same principle as a copying machine or a fax machine. The image is not copied to paper but to an electronic format compatible with paint programs. The biggest disadvantage of a scanner is that it is an extra piece of hardware; thus they are making them relatively expensive compared to visualization programs.

7.7.4

Facsimile Machines

Facsimile machines are essentially specialized scanners. They create an image just as the scanner does and transmit the image across phone lines.

Exercises

7.10 Why do you think scanners produce paint-type pictures rather than draw pictures? (Think about this: you do have all the information you need to answer this question.)

7.11 With your draw or paint program, find the text insertion tool and place a small amount of text into a picture. Now manipulate that text. Try as many of the text manipulation tools provided by your word processor as you can. List which worked and which did not.

(table continued on next page)

7. Believe it or not, *dingbat* is a technical printing term, referring to small designs used to separate segments of text in printed documents.

8. Do not worry about the meanings or derivations of these acronyms. They usually provide little insight into the use of the tool. For example *GIF* means *graphics interchange format* and *JPEG* means *Joint photographic experts group* just not very useful names for most users.

(table continued from previous page)

7.12 If your text processor has drawing or paint tools, try to insert a small drawing into a text you have previously written.

7.8

Computers and Pornography?

One of the truly odd outgrowths of computerized pictures is the growth of the (mostly soft-core) pornography industry. Electronic pictures and collections of pictures are readily available through large data bases or the ads in the back of some magazines. Why is this strange? Mostly because with today's technology, the resolution of computer screens is much lower than that of printed pictures. (In fact, many such pictures are scanned in from existing paper documents.) Nonetheless, there is a distinct subculture that is interested in such pictures.

Is this a bad development? Is it a problem? Many members of Congress think so. Once again your own answer should follow from the first rule of computer ethics. Most aspects of the question really depend on your view of such pictures in general. Placing pictures on a computer seems to change the basic issues very little. If you object to pornography in general, you probably object to pornography on a computer. If you do not object in general, it probably will not bother you in a computerized form. Similarly, your definition of computerized pornography will depend far more on your general definition of the term than it will on the fact that the material is stored in a computer.

There are two exceptions. First, computer systems are often owned by a corporation or institution, rather than by an individual user. Presumably the owners of the system (corporation or educational institution) have some right to dictate how their machines are used. That suggests that not only the user's opinions, but also those of the

system owners, may be relevant. Second, the use of the Internet (see *Chapter 22: Telecomputing*) makes a great many resources both good and bad more widely available. In particular, children may find pornography more accessible through the Internet than through a bookstore. Chapter 22 discusses this aspect in more detail.

Exercise

7.13 Discuss the issue of computer pornography. Take either side on the issue and build an outline of the major points. Then go over the outline and identify which arguments are actually about computerized pornography and which are about pornography in general.

7.9

Summary

Visualization tools help people understand complex ideas by creating visual images. Two classes of visualization tool, paint and draw tools, are based on pixel-and object-level models of images. Each model has advantages and disadvantages. Understanding the models that these tools are based on is essential for their successful use.

Important Ideas

paint drawvisualization

alias pixelobject

clip scanner

art

8

Relations, Tables, and Databases

Knowledge is the small part of ignorance that we arrange and classify.

AMBROSE BIERCE

Pointers

Gateway Databases

Lab:

Lab Unit 7: Using Relations to
Manual: Construct a Database

8.0

Overview

As you have seen, most documents reflect the structure of the data they contain. The hierarchical organization (Chapter 6) is not the only technique for representing more structure than provided by the *ad hoc* techniques (Chapter 5). A *relation* can represent large amounts of data in a uniform manner. Each datum in a relation is represented in an identical manner and with equal weight. This uniformity simplifies the organization and searching of the data collection, and provides the basis for a popular computerized organizational tool: the *database*.

8.1

Uniform Data Structures

A single identifier by itself is seldom very useful. Consider the name:

Bill Clinton

It apparently identifies a person, but tells nothing about him. It

doesn't indicate whether he is real or fictitious, where he lives, or even why we might care about him (even the pronoun is only an assumption so far). At the most, it suggests that there is a person whose name is Bill Clinton. Even the slightly expanded version

Name: Bill Clinton

TABLE 8.1 A List of

Four Names

Bill Clinton

Al Gore

Newt Gingrich

Robert Dole

does no more than make it clear that it is indeed a name. Neither provides any real information. A single datum becomes useful only when combined with additional information.

8.1.1

Lists and Sets

A *list* is a collection of similar data items, such as the names in Table 8.1. A list by itself is no more than a collection of the single data items. The individual items, or *elements*, of the list Bill Clinton, Al Gore, and so on become meaningful only when the list itself is given an *interpretation*. Perhaps the list contains the names of the people invited to dinner next week at my house; perhaps it is the famous "White House enemies" list. Thus each entry in Table 8.2 would represent a fact of the form "Person x is coming to my house for dinner." Notice that the list also implies by omission that Boris Yeltsin is not coming to my house for dinner.

The order of individual items within a list may or may not be important. A list for which the order of the elements is completely unimportant is called an *unordered list* or a *set*. For example, the names in a dinner party guest list are probably listed in an order no more important than the order in which the host happened to think of inviting them. The abstract description of a set as *unordered* confuses some people. It is difficult to visualize a collection as unordered. A printed list (or one displayed on a computer screen), must have some order, if only the first one printed, the second one printed, and so on. For many people, even imagining a list requires imagining it in some order. *Unordered* means only that the order is

unimportant, as in the dinner guest list. Thus, Table 8.3 is a second

TABLE 8.2 People
Coming to My House
for Dinner

Bill Clinton

Al Gore

Newt Gingrich

Robert Dole

TABLE 8.3 People Coming to My House for Dinner

Bill Clinton
Robert Dole
Newt Gingrich
Al Gore

representation of the same set or unordered list described by Table 8.2. A simple unordered list can be used to answer questions of the form:

Does individual x have the property represented by the list?

Was Bill Clinton invited to the dinner party or not?¹ The list actually contains very little informationan item either is or is not an element of the set; it is or is not in the list.

Ordered Lists.

The order of the elements of an *ordered list* does contain information. That information may be:

information about the individual elements of the list, or
information about the list itself.

The list in Table 8.3 seems to be ordered by rank. It tells not only the names of four men, but also who has the most and least seniority. An individual's position in this list indicates his relative rank: Bill Clinton outranks Al Gore, and so on. Thus the closer the person is to the front of this list, the higher his rank (see Table 8.4). In other cases, the order may contain information about the list itself, especially information on how to find an element in the list. The names in Table 8.3 seem to be in alphabetical order. That order tells very little about the individual persons, but it does provide a simple method for determining if the

TABLE 8.4 Our Leaders
(in order of their rank)

Bill Clinton
Al Gore
Robert Dole
Newt Gingrich

1. Compare this to the definition of bit (in Section 5.1, footnote 2) as the basic unit of information.

person is in the list. It is certainly easier to find a name in an alphabetical list than in an unordered list.

Look at each item in the list sequentially:

*If the item is the one you want
then you are done (successfully).*

*If the item is alphabetically later than the one you want
then you are done (unsuccessfully).*

*If the item is alphabetically before the one you want
then you must keep looking.²*

Thus, ordered lists can be used to answer two forms of question:

1. Does individual x have the given property described by the list?
2. How does the individual rank with respect to other individuals on the list (with respect to the ordering property)?

Exercises

8.1 Make an unordered list of the members of your family.

8.2 Order the list from Exercise 8.1 by age. List them again alphabetically.

8.3 Repeat Exercise 8.1 listing the classes you have taken at your university. Now reorder them as in Exercise 8.2, this time in order of the grade you received.

8.1.2

Relations

Even an ordered list of individual elements is of limited use. Including additional properties with each element of the list

Name: Bill Clinton Job: President

makes the list much more valuable or meaningful as a collection of data. Where the name “Bill Clinton” by itself is meaningless, a

combined element can be equivalent to the English sentence “Bill Clinton is the President.” Usually, the structure can be extended in an obvious way as in Table 8.5, which is equivalent to the less compact, but familiar and easily understood representation:

2. In fact, it is even easier than this description suggests. No one searches a phone book this way. Clearly, a phone book can be searched much more quickly. Compare this technique with the one you actually use for searching a telephone book or the method of binary search described in Section 6.4.2.

Bill Clinton is the President.
 Al Gore is the Vice President.
 Newt Gingrich is the Speaker.
 Robert Dole is President pro Tempore.

Computer scientists and mathematicians call such a list a *relation*. Each of the two segments of a relation (or element of the relation) is called a *field*. In the example, “name” and “job” are both fields. Informally, a *relation* is a list of pairs of data items. Each line or element of the relation, called a *record*,³ represents a single entity or real world object.

TABLE 8.5 A “Name and Job” Relation	
Name:Bill Clinton	Job:President
Name:Al Gore	Job:Vice President
Name:Newt Gingrich	Job:Speaker
Name:Robert Dole	Job:President pro Tempore

More Complex Relations.

A relation can have more than two fields. The relations in the previous examples could be expanded to include items such as each individual’s spouse, address, phone number, or year appointed to the office. Each record should be expanded to contain exactly the same list of fields.

8.1.3

Tables

A *table* is simply a compact written representation of a list of records, such as the one in Table 8.6. Each record or element consists of an information pairin this

TABLE 8.6 Table Format	
Representation of Table 8.5	
Name	Job

Bill Clinton	President
Al Gore	Vice President
Newt Gingrich	Speaker
Robert Dole	President pro Tempore

3. Mathematicians and computer scientists use the term *element* even when talking about a relation. The term *record* is used in many computer applications such as databases. Historically, *record* refers to one segment of data from a larger set, such as one paper in a file folder.

TABLE 8.7 Characters and Books
as a Relation

Character	Book title
Tom	<i>The Adventures of Tom Sawyer</i>
Sawyer	<i>Sawyer</i>
Jack Ryan	<i>The Hunt for Red October</i>
Bilbo	<i>The Hobbit</i>
Baggins	
John Gault	<i>Atlas Shrugged</i>
Ishmael	<i>Moby Dick</i>

case one name and one job. Since each record contains the same fields, there is no need to repeat the field name on each line. The two fields of each record are logically tied together. Table 8.7 lists fictional characters and the books in which they appear. We say the relation has five records (Tom Sawyer, and so on) and two fields (character and book title).⁴

Like a set, a relation is an abstract collection of data. A table is a particular representation of that data. The elements of a relation are unordered, but a table is a visual representation of a relation, and therefore has an apparent order. Thus,

Name	Job
Bill Clinton	President
Al Gore	Vice President
Robert Dole	President pro Tempore
Newt Gingrich	Speaker

and

Name	Job
Bill Clinton	President
Robert Dole	President pro

	Tempore
Newt	Speaker
Gingrich	
Al Gore	Vice President

are two different tables representing the same relation. The first table is organized according to the rank of the individual and the second is in alphabetical order. The best organization depends on the needs of the user.

4. Yes, the term *field* is used somewhat ambiguously: this relation has two fields and so does each record of that relation. One can refer to the second field of the relation and to the second element of the record.

8.1.4

Using Relations

The uniformity of the relation structure makes it a powerful tool. Although any list of pairs is technically a relation, the representation becomes useful when each element has the same structure. The relation:

delphinium	blue
America	beautiful
Clinton	President
Ishmael	<i>Moby</i>
	<i>Dick</i>
computer	fun
science	

does not seem particularly useful. In a useful relation, each field has a meaning. The values of two fields in a record indicate a relationship. The name on the right in Table 8.7 identifies a book and the name on the left identifies a character in that same book. Consider two more examples:

Name and phone number.

John Smith	555-
	1234
Mary Jones	123-
	9876
Juan Escobar	321-
	6789

Course and grade.

Computer Science	A
101	

English 102	B
Math 103	C
Art 100	B

In each case, the two parts, or fields, of a record are logically tied together. A list of phone numbers is useless without each corresponding person's name. Grades without courses and prices without products are similarly useless. It is this structure that makes a relation a more useful structure than a simple list. In particular, with a relation or table structure, it makes sense to ask questions of the form:

*Given the value in one field of a record,
what is the value of the other?*

Thus, "What is Bill Clinton's job?" can be restated as

*In the record with name field = 'Bill Clinton',
what is the value of the job field?*

and "Who is president?" becomes

*In the record with job field = 'president'
what is the value of the name field?*

Both questions can be answered based on the relation in Table 8.5. Similarly “What grade did I get in Computer Science 101?” and “In which courses did I get an ‘A’?” can both be answered using the “course and grade” relation. A similar question for records with more than two fields is:

*Given the value in one field of a record
what are the values of the other fields?*

Exercises

8.4 List the elements for each of the following biological relations:

- your sisters
- your parents
- pairs of brothers and sisters in your extended family

8.5 List the elements for each of the following relations:

- pairs of integers less than 20 such that the first is exactly twice the second
- presidents and vice presidents of the U.S. since 1950
- information about courses you have taken
- occupants of dorm rooms
- faculty members in your major department

8.2

The Database:5

A Tool for Representing and Manipulating Relations

Relations provide the basis for many tools, both computerized and otherwise. A phone book consists of pairs: name and phone number, and a catalog consists largely of product names and prices. A course syllabus may list the class-meeting dates for the course and the reading assignments for each date. Generally, tools take advantage of relations in a very specific way:

The user provides the value of one field of a record.

Then the tool provides the other.

A person looks up, or *retrieves*, a phone number in a phone book by providing the name; the phone book provides the number. A student uses the college course schedule by providing the course number, and the schedule tells when the course is offered. Once again, notice that there are two tools involved here: a physical tool,

5. Once again, the general model described here applies to all database programs. The specific details will vary, but the essential capabilities are all the same. Some common brand names of databases include *d-Base*, *FoxBase*, *4th Dimension*, *FileMaker Pro*, and *Oracle*. In addition, most integrated packages, such as *Microsoft Works* and *ClarisWorks* have simple databases which adhere to the same principles.

the phone book, and an abstract tool, the methodology for searching through the phone book.

There are two common computerized tools for specifying tables and relations: *spreadsheets*⁶ and *databases*. In its simplest form, a database is a table or relation. A *database application* is a tool for manipulating that table. Figure 8.1 shows a view of a simple student phone book relation created using a database application. The database is currently listed in alphabetical order. Each record contains information about one specific person: a name and phone number. It could also have held additional information such as student ID number, address, major, and class schedule.

Most students immediately see the value of a word processor. But for many, a database does not seem like a particularly useful tool, at least for personal use. Yet, there are infinitely many potential uses. Consider as candidates any list that you might normally keep on paper. Perhaps you want to keep track of:

Who has borrowed your books.

Your telephone book.

The cards in your baseball card collection.

Useful bibliographic references.

Assignments due in this class.

Things to do for fun.

Your grades.

A list such as this one.

My phone book	
Name	Bill Smith
Phone Number	555-1234
Name	Carmon San Diego
Phone Number	505-7474
Name	Mary Jones
Phone Number	913-9876

Figure 8.1
A Database Representation of a Simple Relation

6. The *spreadsheet*, which takes its name from the corresponding tool used in accounting, is introduced in Section 9.4. Many of the techniques described in this chapter also apply to spreadsheets.

Exercises

8.6 On a database, enter your class schedule with one record for each class. Include the course name, number, instructor, room, and time.

8.7 List five more possible uses for a personal database.

8.8 List five possible uses for a professional database.

8.9 Suppose you have a database of the courses offered at your college or university and that it contains the fields: department name, course number, course title, section number, instructor, time, days, room, and credits. Define queries in the form of Section 8.1.4 for each of the following questions:

What is the course name of Computer Science 101?

What course is Professor Alphonse teaching?

What courses are not offered at 8:00 A.M.?

Are there any courses that are worth five credits?

What courses are in Old Main 101?

8.2.1

Operations on Relations and Tables

Searching is not the only operation people and machines perform on relations. A typical phone book has two basic parts:

the white pages, organized alphabetically by name, and

the yellow pages, organized alphabetically by business type.

The two parts contain very similar information. A person can look up the number of a favorite restaurant in either section. One is easier to use if you know the name, but are not sure if it is primarily a restaurant or a caterer; the other is easier if you can't remember

the exact name (is it *Auberge* or *l'Auberge*?). To many people, maintaining two lists seems wasteful. For example, twice as many trees are cut down to make two versions and twice the printing costs are passed on to the customers. (See Section 8.5.1, Accuracy of databases for a less obvious disadvantage.) A database provides a mechanism for maintaining one set of information but thinking about it as two or more tables. Assuming a collection of data is organized as a relation, what operations are needed in order to view it in multiple ways and to retrieve information easily? Clearly the user must be able to *query* or ask questions of the database. Some common queries or commands follow.

Sort the Relation on One Field.

List the names in alphabetical order. Now list them in order of their age.

Find the Entry with the Highest Value.

What course did I do best in? or just “What was my highest grade?” For many users, the obvious way to find the

record with the largest or smallest value in a field is to sort the database and take the record at the top of the list. The same method could also work for finding all records with a given value in a field. For example, to find all the students who received a B in a course, sort the records by grades and scroll through the list until you find the “B” entries, which will conveniently all be sorted together. This method does have some problems:

It requires that the field of interest contain a measurable or ordered value. The method would not work very well for finding blue-eyed people.

For large databases, the human may still have to scroll through a great many records.

The following commands are usually better for finding specific records.

Find a Record with a Specific Value in a Field.

For example: What is John Smith’s phone number? All databases provide a command, usually called *Select*, for selecting all of the records with a given characteristic: all records with name = bill, all records with eye color = blue, and so on.

Find All Records with a Specific Value in a Field.

What are the class times for each section of *Computer Science 101*? Notice that in terms of relations (and consequently for databases), these first two questions are actually identical. The wording itself seems to presuppose that one or more than one record satisfies the question. The definition of relation makes no such presupposition.

Count the Number of Entries with a Given Value.

How many sections of *Computer Science 101* are offered?

Some additional tools for manipulating records are discussed in Section 18.4.3.

8.2.2

Analogy with Other Tools

While the operations based on relations may seem new, you can still learn to use a database by drawing on your experience with other computerized tools. A database application is not a totally new tool, unrelated to the ones you already know. Your existing knowledge tells you much about related concepts in this new tool:

Insertion point: as with a word processor, each operation assumes an operand.

Selection of a region: with a word processor you can select a regionseveral characters, words, or sentences. The same must be true of a database, and the same techniques will work.

Delete: word processors allow you to erase your mistakes; database tools allow you to delete entire records or individual characters.

Cut/copy/paste are the stock and trade of the word processor.
All are possible in a database.

File operations: every application must manipulate its data files. The database methods are the same for all common operations such as saving, printing, making, and opening files.

Formatting documents: changing the width, scrolling the windows, and so on.

Characters: come in various fonts and sizes.

Use of keyboard and mouse: even the interface is the same.

Exercises

8.10 Suppose the order of a database includes information. For example, a class list presented in order from highest to lowest cumulative score implies that a person's position shows her class rank. What happens if the user of a database sorts the list into alphabetical order to make it easier to find a specific student?

8.11 Phone books come in both white and yellow varieties. Can you define another possible order for a phone book? Can you think of a use for that ordering?

8.3

Living Documents

A database document or at least its use is fundamentally different from other document types discussed previously, such as those associated with a word processor, outliner, or drawing tool. Like the previous tools, a database tool is a tool for creating a document. However, most other documents with which you are familiar are essentially static. The user uses the application to create a document. Once created, most documents are printed (or otherwise displayed) for delivery to a final destination and there the use of the application is complete. For example, a student writes a term paper,

prints it, and turns it in; the instructor does not use a word processor to read the paper. The document may be updated later (for example, a student writes the draft of a paper, gets some feedback, and rewrites it in its final form), but for most purposes once a document is complete, the application and the document can be separated.

A database, in contrast, will probably have an ongoing and changing life. For example, suppose the phone book in example in Figure 8.1 is a personal phone book listing the friends of one particular student. Every time the owner of the database makes a new friend, the database needs updating. If you collected matchboxes and kept a catalog describing all of the items in your collection, you would want to update the database every time you added a new matchbox to the collection. Database documents are usually tools with an extended and evolving lifetime.

So far, a word processor could provide the same service. A user could keep all of the entries in a word-processor document. Whenever the list of friends grew, shrank, or otherwise changed, the user could modify the list and print a new

Box 8.1 The Advantage of Reuse

Far too often, new users are reluctant to use a computer for a given task. "It will take me longer than doing it by hand!" they say. In fact, this is often a true observation but a short-sighted one. Computerizing a task brings with it two significant time savings. First, the *learning curve*—the time it takes to learn a new task—predicts that subsequent use of an application will require less time. The more you use a tool, the less time it takes. Using an application for a task, even if not essential, moves you further along that curve. Soon you will find that the tool does in fact save you time. Unfortunately, you will never realize that savings unless you start.

Second, once you create a new tool, you can use that tool again. The second time you use the tool, you need not create it from scratch, but you can reuse your earlier work. It is not necessary to recreate the database describing your wine cellar each time you use it or update it.

listing. But the database application provides an even more fundamental service than updating: *data retrieval*. The primary reason for keeping a collection of data in a database is fast access to that information. The database tool can find any item or group of items quickly. This implies that the database application should be part of that retrieval process. Similarly, in order to maintain multiple views of the same data, the application must be part of the process. To take advantage of a database, the owner needs continued access to the database tool. The user does not want to look items up on a sheet of paper. Instead she wants to ask the database application to look them up.

8.3.1

The Document Life Cycle: Designers and Users

The *life cycle* of the database can be divided into (at least) three major phases:

1. design,
2. entry, and
3. use.

The person who actually performs the first set of operations is called the *designer* and the one who performs the third set is the *user*. The designer and user may or may not be the same person. For example, the designer and user of a telephone

book are different people, but the designer and user of a personal inventory (e.g., listing of baseball cards) will often be the same person. Before entering any information, the designer must decide what items will go in the database, the names of the fields, how the individual fields will be displayed, what values are legal, and so on. After a database is complete, the user will want to access the data. The person who enters the data may be either the designer or user. For example, the designer may enter the initial data, but the user updates the data as it changes. The remainder of this text uses the terms *designer* and *user* separately to distinguish between the two roles, even in contexts where they are likely to be the same person.

The Design Phase.

During the design phase, the designer must first define the relation on which the database will be based, including what fields are needed to provide all the information that may be needed. The designer should name the fields and define how they will be arranged on the screen (a *layout*). Note that the designer creates the logical parts of the records, but does not place values into the individual fields or even determine how many records there will be. For example, the designer may decide that each record will contain fields for name, address, phone number, and major, but no actual data is entered. Since the designer is creating new objects (the fields), the operations of this stage may seem more analogous to a draw tool than to a word processing tool: create a new object, position it, name it, and define its appearance.

Do not neglect this stage. The design stage may seem trivial. Relations in student databases may contain only a handful of fields. But careful attention to this stage will make the other stages much easier.

The Data Entry Phase.

Once the database is designed, each field of each record must be

filled in. This stage requires two types of action:
create a new record, and
place values in the fields of the new record.

The designer must find and enter the data, filling in each field of each record in the relation. Most database applications allow missing data, but of course there may be no way to find a record with missing data in a given field. The operations at this stage are most reminiscent of a word processor: the data entry person is simply typing new data (although within a constrained environment).

The Use Phase.

Once a database is created and the data entered, it is available for use. The user can look up individual records, sort the entire relation into any order, find highest and lowest (first and last) records in a given order, and find records with ranges of values (tests for which the student received a grade of 90 or higher). In addition, the user can add or correct individual entries. Most

applications allow the user to generate reports (e.g., list all of your classes by semester, with the GPA calculated for each semester), or even mailing lists. The operations at this stage may seem reminiscent of those of an outline tool, with the granularity between that of the other two stages.

8.4

Using a Database Tool

8.4.1

Granularity

Once again, granularity helps define the basic capabilities of the tool: record operations. Generally, the operand of any operation is a single record. For example, “Find the student with the highest grade” should locate one record. Some operations work on fields, particularly those used by the designer: “Create a field for this relation called ‘address’.” It is possible, but not convenient, to work at a lower level by selecting the record or field and then performing operations on the individual characters within the field. A database’s capabilities at the character level are minimalcertainly less than most word processors. For example, you may not be able to change the style or font of individual characters. Normally the format of every record must be identical.

8.4.2

Techniques for Describing Relations

As with any other tool, there are good ways and bad ways to use a database application. The following suggestions are guidelines for good and productive use of databases.

Remember the User.

That is, the general format is set up by the designer, but a different person enters and uses the data. Clearly the user will want to know which item is which. The designer must make the definitions of

fields clear through formatting and the use of instructions for the user.

Define for Identification.

Each record should be identifiable. For example, suppose you have a personal address book which contains two entries for “Bill.” The user must be able to identify which entry corresponds to which real world individual. Often this requires additional fields (e.g., for family name or other identifying characteristics).

Do Not Double Use Fields.

Suppose you have two fields for which most entries have no useful value. For example, a field for personal fax number, or the name of a person’s tenth child. Many designers are tempted to double use this field. That is, they will enter the child’s name if there is one, but otherwise leave the field available for that extra phone number. This will only lead to confusion. For example, a user seeking a fax number may need to ask “Tell me the name of the tenth child.”

Be Complete.

Provide fields for all data items that you expect to need in the database.

Format for Clarity.

No one likes to wade through irrelevant or poorly structured documents to find the single important piece of information. Have you ever missed the “fine print” on a contract? Organize the document to help the reader or user. Format the individual records wisely. Make sure each field is large enough to display any value it may hold.

8.4.3

More Complex Situations

Databases may be far more complex than this chapter may seem to imply. For example, a database may be composed of several distinct but interconnected relations. Or an application may support queries of the form:

*Find all the records with
‘computer science’ in the department field and
a value greater than 200 but less than 300 in the course
number
field.*

Unfortunately, the database tools available to students in introductory courses often have very limited capabilities. *Chapter 18: Applied Logic* addresses some of these issues.

Exercises

8.12 Many people have trouble budgeting or even keeping track of their money. Some “money professionals” recommend the following technique for getting started. Write down everything you spend for two months. At the end of that time,

go through and classify each item as essential, optional, or “needed, but could be reduced.” Add up the amounts in each category. Create a database version of this and record every item faithfully for two weeks.

8.13 List five more possible uses for a database beyond those that you thought of when first asked to do so in Exercise 8.7.

8.5

Ethics of Data Collection

Databases create an interesting set of ethical questions that may not be answerable by the first ethical rule of computing. These questions are not so much issues of what you should do, but of what others are doing.

Some tasks simply are not feasible without a computer. Since they are not feasible, no one worries about them too much. For example, suppose the Internal Revenue Service had access to all of your bank records, all of your payroll records, and all of your credit card records. In short, they had access to records of

your entire financial status. It might be possible for them to sort through exactly how much you spent on any given item. From this they could easily spot inconsistencies. For example, if you claim to have two dependent children but do not spend any money on children's clothes, then something must be wrong. Possibly, you can explain this: perhaps your children inherit all of the clothes from their older cousins. But you would not find the process to be much fun.

Of course, the IRS has the right to search for tax fraud. So that example may not bother many people. Consider a similar example: suppose the local stores can keep track of all your purchases. They already do this to some extent: electronic scanners record the specific purchase, and if you use a credit card, discount card, or check cashing card, the store has your name. The store could then use the list of your purchases to generate targeted advertising. That is, they could send you ads based on what you seem to purchase most often. Or they could sell the information about you to other companies. Have you ever purchased anything that you would prefer others not to know about? Many things are not illegal, but you would just prefer that they be private. For example, have you purchased medication for illness of a private nature? Do you purchase more alcohol than you want to admit? Are you embarrassed by the brands that you purchase (are they too expensive, too cheap)?

The bottom line is that large database tools make it possible to collect very large sets of data. The possessors of those large sets of data can determine much about you and your habits. This creates an interesting question: Who owns information about you? Is it yours, or does it belong to whoever possesses the database? If the latter, are there any restrictions on what that person may do with the data that describes you and your habits?

8.5.1

Accuracy of Databases

The creation of any collection of data introduces the possibility of incorrect data. Any such inaccuracy is particularly problematic when it involves personal information. For example, several companies maintain very large databases containing credit information about almost everyone. The database contains “private” information such as the amount an individual has borrowed, whether the money was paid back, how often payments were late, how long the person has been employed, and even health records. When the data is inaccurate, individuals may be denied credit, based on the faulty information. In spite of the issues mentioned earlier in this section, maintenance of data in a database offers several advantages over other methods:

Updating information: a computerized database can be updated more quickly and is therefore more likely to be current.

Single sources: a single database that can be sorted into any order provides fewer opportunities for error than would any data collection with separate copies of the data in each of the needed orders.

Exercises

- 8.14 Write an essay balancing the individual's right to privacy against the right to gather information. Be sure to address the issue in the context of computerized information.
- 8.15 Suppose a private agency collects information about you, but that information is in error. And suppose that the information causes you harm (e.g., you are denied credit). What compensation should you be entitled to?

8.6

Summary

A relation is an unordered collection of records, each describing a single entity. The record, or single element of a relation, is made up of fields, each describing a particular aspect of the element. A database application is a tool for manipulating relations: it can save, reorganize, and search collections of data represented as relations. A database is a living document; it continues to change over time and the application continues to work with it.

Important Ideas

set list ordered list

record relationdatabase

field sort search

element user designer

query select living
document

9

Functions and Calculated Values

Form ever follows function.

LOUIS HENRY SULLIVAN

Pointers

Gateway Expressions

Lab:

9.0

Overview

Chapter 8 demonstrated the usefulness of relations for organizing information. Although zero, one, or more records in a database may have a given value in a given field, relations for which the values in one field are uniquely determined by values in other fields are particularly interesting. This chapter introduces a special subset of relations, called *functions*, and explores their use for building collections of related data. Unlike arbitrary relations, functions can guarantee unique answers. The chapter also introduces a computerized tool for creating and managing functional data.

9.1

Introduction to Functions

For some data collections, uniqueness is especially important. In a data collection organized as a relation, a user might seek a single specific record, as in:

Who is the manufacturer of your computer?

The answer

It might be IBM and it might be Apple.

is unlikely to be satisfactory. One single company manufactured the computer. Thus, a spreadsheet with two very similar records:

owner	serial number	manufacturer
Greg	12345	IBM
Greg	12345	Apple

TABLE 9.1 A Menu with Prices

Hamburger	\$2.95
Cheeseburger	\$3.45
Fries	\$.90
Onion Rings	\$1.45
Coke	\$.75
Coffee	\$.50
Pie	\$.90

seems quite unsatisfactory. Similarly, the registrar's office at any college maintains a collection of data for every student. Using the "student ID" field, the registrar can retrieve the data record for any single student. Two students with the same ID would certainly confuse the system.

A relation in which the value in one field identifies a unique element or record is called a *function*. The restaurant menu in Table 9.1 is a function. It is clearly a relation: a set of product-price pairs. Each record contains the name and the price of a product. In each case, the product name uniquely identifies a single record in the relation. A menu in which one hamburger had two prices would be disconcerting.

In the relation in Table 9.2, the title of each book uniquely identifies the author.¹ The relation is therefore a function. Because a value in one field uniquely identifies an element or record, it also uniquely identifies a value for the other field(s). In this case, each book title identifies a unique author. Other possible functions include:

Student	↳ student's mother
Student ID	↳ student
Husband	↳ wife
Wife	↳ husband
Company name	↳ chief executive officer
College	↳ state it is located in

TABLE 9.2 Authors and Books

Samuel	<i>The Adventures of Tom</i>
Clemens	<i>Sawyer</i>
Tom Clancy	<i>The Hunt for Red October</i>
J.R.R.	<i>The Hobbit</i>
Tolkien	
Ayn Rand	<i>Atlas Shrugged</i>
Herman	<i>Moby Dick</i>
Melville	
Tom Clancy	<i>The Sum of All Fears</i>

1. Yes, some books have more than one author. And that could mean that the relation is not a function. Assume for now that the author is “Smith & Jones,” rather than treating it as two records, one listing an author as “Smith” and one listing it as “Jones.” After all, Smith did not write the book by herself, so any record containing only her name would be misleading.

In each of these pairs, a given value on the left² uniquely identifies a record containing that value, and therefore, a single right-hand field (ignoring the possible but highly unusual cases such as a college that is actually located in two states, or a man who is married to two women at the same time). Notice that the definition of function only requires that there be at least one field that uniquely identifies records. It does not require that both fields do so. A function in which both fields uniquely identify records is called a *one-to-one* function. The preceding functions describing husband and wife are apparently one-to-one functions. We say that those two functions are *inverses* of each other.

9.2

Computability of Functions

A large subset of functions possess a second important property: one field of a record can be calculated automatically from information in other fields of the same record. For example, one way to create a relation representing employees' pay is to create each record manually as:

employee	rate	hours	total pay
Bill	\$5	40	\$200
Mary	\$6	35	\$210
Juan	\$7	30	\$210

But, the total pay should be uniquely derivable from the hours worked and the hourly rate of pay. An alternative representation of a function uses instructions (to a person or a computer) telling how to *calculate* or *compute* each needed value. In this case, the table could be written:

employee	rate	hours	total pay
Bill	\$5	40	$\$5 \times 40$
Mary	\$6	35	$\$6 \times 35$
Juan	\$7	30	$\$7 \times 30$

Notice that the method of calculating the right-most field is exactly the same for each person:

`total pay = rate × hours`

2. There is nothing inherently special about the choice of the left columnthese examples were written this way for ease of discussion. Traditionally, the field that defines the record goes on the left hand side.

Functions provide a means for exploiting such commonalities within a relation. Table 9.3 shows a function or relation describing the first ten natural numbers³ and their squares. The single equation:

$$\text{square of } n = n \times n \quad (\text{EQ 9.1})$$

describes exactly the same function. In fact, EQ 9.1 contains much more information than the table, because it also defines how to find the squares of 11, 1000, -5, and 2.7. The designer need not explicitly list every possible record, or calculate new values as they are needed.

Rather than explicitly building every field of every record, the designer of a database can ask a computer to calculate some of the fields. Not every function can be computed. For example, given two competitors in a tournament, there is no way to automatically fill in the winner's slot (indeed tournaments would be quite dull to watch if we could do so). If the relation in Table 8.7 also contained a record for *The Three Musketeers*, no automatic function could add D'Artagnan to the character field. A user who happened to know the names of the characters could add them but in general they cannot be derived from the name of the book.

Extending the Menu.

Suppose the management of a restaurant (believing in truth in advertising) wanted to inform their customers not only of the base prices of each item, but also the price including tax (which is, after all, what the customer really wants to know). They could expand their menu (from Table 9.1) to:

Item	Cost	Cost with Tax
Hamburger	\$2.95	\$3.16
Cheeseburger	\$3.45	\$3.70
Fries	\$.95	\$1.02
Onion Rings	\$1.45	\$1.56

Coke	\$.75	\$.80
Coffee	\$.50	\$.54
Pie	\$.90	\$.96

Rather than calculate every value in this way, most restaurants simply tell their customers that “7% sales tax will be added to all purchases,” which does not explicitly provide the answers the customer needs. Alternatively, they could instruct a computer to calculate the values.

3. The *natural* numbers are just the positive integers. Exercise 16.7 provides an example of their use that illustrates why the name was chosen.

TABLE Squares of
9.3 Natural
Numbers

1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81
10	100

The most obvious class of *computable* functions is the arithmetic or mathematically defined functions.⁴ For example:

Addition is a function from two values (the *addends*) to one value (the *sum*).

Multiplication is a function from two *multiplicands* to a *product*.

But functions are not restricted to mathematics. Many functions that do not seem at all mathematical or arithmetic can also be "calculated." The second and third fields of the relation:

letter predecessor successor

...

d	c	e
e	d	f
f	e	g

...

could be calculated or described as:

predecessor = the letter before the given letter (in alphabetical order)

successor = the letter after the given letter (in alphabetical order).

Often the calculation can be performed mechanically so mechanically that a computer can do it. We refer to this process as a calculation even though there does not seem to be any mathematics. The set of computable functions includes many of this form.

4. At this point some readers may complain, saying “I can’t do math! Don’t make me!” or “This wasn’t supposed to be a math class!” Don’t worry for two reasons. First, there will not be any complicated math (well, not very much; and you will be guided through any tough parts). Second (and even more important) math is not a thing to be afraid of any more than any other academic subject area. (Can you imagine someone saying “I can’t do reading. Don’t make me!”?) You might also discover an exciting side benefit: no one need be intimidated by math. A computer can take care of the drudgery aspects of math and reduce the number of silly errors that we all make. In fact, you will hopefully see math as a powerful tool to be employed for your benefit.

Box 9.1 Computability

Computable is actually a formal term in computer science, with a much more rigorous definition than the everyday language definition used in this chapter. Roughly, a function is said to be computable if it is possible to specify the calculation to a computer. But we require that the calculation will always work and will always do so in a finite amount of time. The real surprise is that it is not always possible to meet these additional requirements. This doesn't mean simply that a particular individual fails to find the method, but that no one could ever find such a method, no matter how carefully they worked or how fast a computer they had.

The incomputability of some functions follows directly from the definitions; for example, any attempt to calculate the decimal representation of $1/3$ is doomed to failure because the answer: $0.333333\dots$ contains an infinite series of 3s. It would be impossible to list them all in finite time. Any attempt to find the exact decimal representation of π (pi) meets the same fate, as would any attempt to list all of the possible integers. The incomputability of other functions can be much less obvious. The identification of incomputable functions has played a central role in the field of computer science.

Exercises

9.1 Which of the following biological relations are functions?

- sister of . . .
- aunt of . . .
- mother of . . .
- daughter of . . .
- eldest daughter of . . .
- grandmother of . . .

- 9.2 Which of the following nonbiological relations are functions?
- instructor of a course.
 - president of the U.S. on a given date.
 - height of a person.
 - month in which the n th day of the year occurs.

(table continued on next page)

(table continued from previous page)

midpoint of a line.

result of a chemical reaction.

9.3 Calculate, derive, or determine the results for the following functions:

The sum of the costs of all items in the menu of Table 9.1.

The month portion of today's date.

The date exactly one week ago.

The day of the week on which your graduation will be held.

The name after yours in the phone book.

Your name written with no upper case letters.

9.4 For each function in Exercises 9.1 through 9.3, state whether or not you believe a computer would be useful for finding the answer.

9.3

Reasons for Using Functions

Functions have many advantages over simpler relations. Consider an example:

Three students rent an apartment for a total
of \$900 total (or \$300 each) per month.

This statement contains three separate, but related, pieces of information:

There are three students.

The total rent is \$900.

The cost to each student is \$300.

Any two of these facts provide all of the needed data. The designer can calculate the remaining fact from the others. Better yet, she can

ask a computer to do so if she can describe the relationship between the two known values and the *unknown* value. This requires bringing a little knowledge of the world to bear on the problem: in this case, the three students would usually each pay equal shares: one third of the total. One technique organizes the information as in Chapter 8:

number of 3

persons:

total rent: \$900

rent per person: \$300

which does not take advantage of the mathematical relationship that exists between the fields. An alternative is to write:

number of 3

persons:

total rent: \$900

rent per person: total rent

3

and ask a computer to calculate the last value. Ideally, the designer would like to specify the least possible data and let the computer calculate the remaining items. Doing so:

saves the time required to calculate the rent per person,

decreases the likelihood of error, and

provides a mechanism for dealing with future changes.

Consider these advantages one at a time.

9.3.1

Save Time

Saving time may seem like the lazy man's approach. Calculating the rent per person doesn't tax many people very much. But if the problem were just slightly different perhaps seven roommates and a total rent of \$1,342.69 it would be significantly harder. Additional complications perhaps the house had two bedrooms and the rent is differentiated, with the person who got the single room paying more would make it even worse.

9.3.2

Reduce Errors

Decreasing the likelihood of error not only improves the quality of any work, but it reduces the time required to get the task done.

Since computers do not make arithmetic errors,⁵ computerizing the task reduces the likelihood of error. Similarly, any relation that can be described once rather than listing many values reduces the opportunities for error. Even a simple error increases the total time to complete a project. If the three roommates above wrote out their checks for \$200 each, they would have to rewrite the checks.

9.3.3

Deal with Change

The most significant advantage of functional specification comes in

the form of a “delayed gratification”: it will not be realized until the data changes. But the data will change, that is a given. Change, in many forms, is virtually guaranteed:

Predictable Changes.

Suppose next month’s rent increases to \$950. If the definition of the “rent per person” remains constant, the user need only change the total rent and ask a computer to calculate any resultant changes. Similarly if you bring in a fourth roommate, you can again ask a computer to recalculate the rent per person. Some changes are completely predictable, like changes in the date.

5. Although you have often heard people blame problems on “computer errors,” true computer errors are extremely rare. Most “computer errors” are no more the fault of a computer than “typos” or “typographical errors” are the fault of the typewriter.

Some are partially predictable. We know the rent will increase; we just don't know when, or by how much.

Errors.

Some changes result from errors. If, for example, you wrote \$90 rather than \$900, or read the bill as \$999, you must recalculate the value using the correct total rent. A correctly specified function allows such an error to be fixed by changing only the incorrect entry—the function recalculates the rent per person. Unfortunately, errors are a fact of life in the computing world. Every computer user spends significant amounts of time correcting his or her own errors. The use of functions helps reduce that effort.

Reuse.

Once you have built a rent calculator, your friends may want to use it for their own rent. But they may have a different number of roommates or different total rent. The *reuse* of tools is an essential component of most computer use. The fact that a tool can be used again is what makes it worth the time to use it once.

9.3.4

Terminology for Functions

Functions are an essential tool of many academic fields. From your work in other areas, you may already be familiar with the terminology of functions, or perhaps you are seeing it now as a prelude to work that is yet to come. In the examples above, some fields were specified as raw data, and others were calculated. In formal terms, the raw data is called the *independent values* or *independent variables*, and the calculated values are called *dependent values* or *dependent variables*. Independent values are generally input from an external source. For example, the rent is the same no matter how many students share the apartment, a number which must be defined before calculating the rent per person. The dependent variables can often be calculated from the independent

variables. In this case, the rent per person is dependent on both the total rent and the number of persons. These values are called *variables* because they can vary or take on different values without changing the relation or function definition.

Writing functional descriptions transforms the task from

Write down all the data.

(Perhaps, calculate some of the values yourself.)

to

Write down the raw or independent data.

Provide the functions defining the dependent data.

Think of a function as an expression that describes the relationship between two values. In particular it should describe how to calculate the dependent values *from* the *arguments* or independent values. We describe a function as *from* the independent *to* the dependent variables. For example, the rent calculation is a function *from* total-rent and number-of-persons *to* rent-per-person .

The process of calculating one value from another is so common that the process itself is often referred to as a function usually with the same name as the relation or a field in the relation. Given the statement: “The author of *Moby Dick* is Herman Melville,” which appears as one element of a relation in Table 9.2 as:

author	title
Herman Melville	<i>Moby Dick</i>

one might say the author is a *function of* the title, or that Herman Melville is the *value of* the function. Such function definitions are sometimes written formally as:

$$\text{person} = \text{author}(\text{book})$$

or

$$\text{author: books} \rightarrow \text{persons}$$

Exercises

9.5 Give the independent and dependent variables for each function in Exercises 9.1 and 9.3.

9.6 Calculate or derive the results (value of the dependent variables) for each of the following nonmathematical functions and independent variables:

The president after (Carter).

The first ten characters of this sentence.

The last ten characters of the previous sentence.

The result of converting all characters of this sentence to upper case.

The month portion of the date: 1 September 1995.

9.7 Calculate or derive the results (value of the dependent

variables) for each of the following mathematical functions and independent variables:

The $\sin(p)$.

The midpoint of the line between the points (2,3) and (6,6).

25.

210.

The cube root of 216.

9.8 Exercise 9.1 has some implications for hierarchically structured data. Compare the results of that exercise with the general properties of trees.

6. For the mathematicians: yes, this is the same notation used in calculus and other math courses. Formally, mathematicians say that books is the *domain* of the function authors and persons is the *range*.

9.4

A Tool for Representing Functions: The Spreadsheet

The most common tool for representing functions is a *spreadsheet*.⁷ The spreadsheet takes its name from the corresponding tool, long used in the business world for purposes such as bookkeeping. An accountant, who needs to balance the books, first enters all the important details on a spreadsheet. Like a database, a spreadsheet is a tool for organizing information. In any spreadsheet, the user may enter:

values (raw data),

labels⁸ that help identify the values, and

functions that define values based on other cells.

The lastthe spreadsheet's ability to *assign* values to a cell based on a calculationis its basic strength. Just as the form of Lewis's architectural creations (see opening quote) followed from their function, the form of a spreadsheet follows from its use. Visually, it appears as a *matrix*, or grid of *cells*, as in Figure 9.1. The user can enter values (such as \$900 and 3) or labels or other written information (e.g., the words: "total rent") into any cell. The cells are

Data entry area				
	A	B	C	D
1				
2		number of persons		3
3		total rent:		\$900.00
4		rent per person		\$300.00

Figure 9.1

A Simple Spreadsheet

7. Although many spreadsheets exist, they are all essentially the same in terms of their basic operation. Some common commercial names of spreadsheets include *Excel*, and *Lotus 1-2-3*, as well as spreadsheets contained in the popular integrated systems such as *ClarisWorks*, *Microsoft Works*, and *Great Works*.
8. Many students have experienced a math or science instructor who seemed fixated on “units.” “You must specify the units or the answer is wrong!” she might say. Labels are roughly the computer science equivalent. Specifying the units or definitions as labels will save you much future grief. For now, specify the label or the units in one cell and the quantities in another.

organized by *rows* and *columns*. Traditionally, columns are identified by letters of the alphabet, and rows with numbers. A row number and a column letter uniquely identify one cell location. To refer to a value, simply refer to the cell in which it resides. In the figure, the value “\$300” is in cell D4 .

Exercise

9.9 Repeat Exercise 8.6, but this time using a spreadsheet: enter your class schedule with one row for each class. In that row, place the course name, number, instructor, room, and time.

A spreadsheet is in many ways similar to a database application. Both applications are living documents capable of representing both the structure and organization of relational information, but a spreadsheet application is specifically designed to represent and take advantage of functions.⁹ The user may specify a function definition in a cell, telling the spreadsheet to calculate the value of that function. Any cell whose definition refers to independent values must identify those independent values. In a database or when solving algebra problems, variables or fields are called by name: total rent or number of persons . But a spreadsheet refers to values by their cell locations. Thus, Figure 9.2 does not refer explicitly to \$900 or total rent but to D3 , because the cell at row 3 of column D contains the value representing total rent . The original equation:

$$\text{rent per person} = \text{total rent}/\text{number of persons}$$

is rewritten as

$$D4 = D3/D2$$

The equation, or *assignment*, itself goes in cell D4 . Notice that the left side of the expression describes the value in cell D4 (the cell where the equation is stored). In

	A	B	C	D
1				
2		number of persons		3
3		total rent:		\$900.00
4		rent per person		=D3/D2

Figure 9.2
Simple Function in a Spreadsheet

9. Individual spreadsheets and databases will vary. Most spreadsheets offer some capabilities that are more characteristic of databases and *vice versa*. But in general, the characteristic differences are characterized by the focus on relations in general or functions specifically.

fact, the left side of every such function identifies the cell containing the function. Since it is repetitious, the cell name is usually omitted:¹⁰

= D3/D2 |

9.4.1

Historical Anachronisms and the Representation of Functions

The available methods for specifying functions in spreadsheets have some annoying limitations. Some limitations are actually arcane relics of a bygone era. Others represent current practical problems. The available character set limits the possible representations. In the noncomputerized world, the most common arithmetic operations are normally denoted by standard symbols or representations:

$a +$ the sum of a and b

b

$a - b$ the difference between a and

b

$a \times$ the product of a and b

b

\underline{a} the quotient a divided by b ,

b and

ab a raised to the b power

In spite of recently released computers, such as the Apple *Newton* that can read handwritten material, few computers can yet handle all of these conventions.

Multiplication.

Mathematicians have long used “ x ” as a favorite name for a variable, or an unknown value. But few typewriters or computer keyboards have two distinct keys for “ x ” (the 24th letter of the alphabet) and “ \times ” (multiplication sign). Since both uses do occur in many applications, using the same symbol for both purposes can be

ambiguous. Does

ax**b**

refer to a single quantity called “ $a \times b$ ”, or to the operation “ a times b ”? By convention, virtually all computer applications now use “ $*$ ” rather than “ \times ” for multiplication:

a***b**

Similarly, mathematicians do not always indicate multiplication explicitly, specifying it by juxtaposing the multiplicands (e.g., AB , rather than $A \times B$). This is ambiguous in many computer applications (e.g., is “ AB ” the product of A and B , or the two-letter name of a single quantity?). By convention, most computer

10. Notice that Figures 9.1 and 9.2 are two views of the same spreadsheet. One shows the function description and the other shows the numeric value. All spreadsheet applications allow the designer to choose and change the format of the display.

3	cost of the food	10.13
4	standard tipping rate	15
5	amount of tip	=D3*D4/100

Figure 9.3
Representing Multiplication and Division

applications also require that the user explicitly indicate all operations including the multiplication operator.¹¹ Division.

Division has a similar problem. The traditional representation for division, $\frac{a}{b}$ uses a vertical organization: the dividend above the divisor. Typewriters and simple text processors use a linear organization. There is no mechanism for grouping characters vertically. In most computer applications, the user specifies “ a divided by b ,” by placing the dividend before the divisor:

a/b , rather than $\frac{a}{b}$

Figure 9.3 shows a spreadsheet representation of a function for calculating a waiter’s tip, which includes both a multiplication and a division.

Exponentiation.

Exponentiation is popularly called “raising to a power” because the power is represented as a superscript. 4^3 means, roughly, the product of three 4s: $4 \times 4 \times 4$ (or perhaps we should say: $4*4*4$).¹² Since older computers had no way of entering a superscript, an alternative was needed. The accepted convention uses the *circumflex* (“ \wedge ” or SHIFT-6 on a keyboard) to indicate raising to a power. Indeed, the circumflex reminds many users of an upward-pointing arrow, which in turn, suggests “raising.” Thus a^b means ab , as in the calculations for Table 9.3 shown in the spreadsheet segments in Figure 9.4.

11. Actually, the two problems involving multiplication are seldom an issue in spreadsheets. But the “*” operator is so universally accepted that all spreadsheets have adopted the convention. One possible problem occurs in large spreadsheets. Any spreadsheet with more than 26 columns needs a convention for naming the later columns. “AA” is the common name of the column after column “Z”. So a spreadsheet with 50 columns has a column $A \times 2$, and a cell $A \times 3$.

Also notice that computers with numeric keypads (section to the right of main keyboard containing numbers and arithmetic operations) have an “*” rather than an “ \times ” in the keypad.

12. Notice the difference between the wording in this definition and the popular, but incorrect wording: “ a multiplied by itself b times.” Four multiplied by itself 3 times, would require three multiplications: $4 \times 4 \times 4 \times 4$. This common misstatement comes back to haunt many students when they attempt to calculate exponential values.

	A	B	C
1			
2		table of squares	
3		number	square
4		1	1
5		2	4
6		3	9
7		4	16
8		5	25
9		6	36
10		7	49
11		8	64
12		9	81
13		10	100

	B	C
1		
2		table of squares
3		number
4		square
5	1	=B4^2
6	2	=B5^2
7	3	=B6^2
8	4	=B7^2
9	5	=B8^2
10	6	=B9^2
11	7	=B10^2
12	8	=B11^2
13	9	=B12^2
14	10	=B13^2

Figure 9.4
Table of Squares

9.4.2

Expanding the Basic Set of Operations

In addition to the standard arithmetic operators, spreadsheets provide many more complicated tools, called *built-in functions*, because the definitions of the functions are built into the spreadsheet application; the user does not have to define them. Many serve special purposes such as financial calculations, or representation of date and time information. Table 9.4 lists several examples.¹³ You can find out about many others from the user manual, help programs, or the Edit menu for your spreadsheet.

TABLE 9.4 Some Examples of Functions

Function	Description	Value
max (5, 3, 9)	The maximum value among the arguments	9
min (5, 3, 9)	The minimum value among the arguments	3
mod (7, 4)	The remainder after dividing 1st argument by 2nd	3
upper ("abcd")	Converts all arguments to upper case	ABCD
average (5, 7)	The average of all arguments	6
proper (john doe)	Makes the argument a proper name	John Doe

row (A3)	The number of the row containing the argument	3
dayname (3)	The name of the <i>n</i> th day of the week	Tuesday

13. The exact name of functions will vary from spreadsheet application to application. But all will have functions with similar names that perform similar calculations. The best source of the exact names is the application itself.

Specifying Function Names.

Although any well-specified function can be implemented on a computer, relatively few (perhaps 100) are implemented in a given spreadsheet application. Even fewer functions have standard symbolic representations. One reason for this is that there are not enough special characters (e.g., +, -, *, /, ^) on the keyboard (what symbol represents "average"?). A second reason is the number of operands for a function. All functions in Section 9.3 are *binary functions*. That is, each function has two arguments or operands and yields a single answer; addition adds two numbers, yielding a sum; subtraction subtracts one number from another, yielding a difference, and so on. The conventional notation, called *infix* representation, is ideally suited for binary operations. The symbol for the operation sits in between the two operands. There is no analogous format for operations with a greater number of operands. Most other functions are represented in *prefix* notation: the mnemonic name (e.g., max, min, average, or sum) is written first, followed by a list of the operands, grouped together within parentheses,¹⁴ as in Table 9.4. Figure 9.5 shows several more functions the way they actually appear as used in a typical spreadsheet.

General Expressions.

In general, you can specify any well-defined function. The critical phrase here is "well-defined." If you cannot *describe* the needed computation, there is no way to ask a computer to do it. The computation may be simple or complex, but it must be unambiguous and never fuzzy. For example nicest cannot replace max in Table 9.4, because the definition of *nicest* varies from person to person.

9.4.3

Examples of Uses of Spreadsheets

While most students immediately see the value of a word processor, for many a spreadsheet seems more like a tool for an accountant than a student. Yet there are many uses for even the most casual user, for example:

- Balance your checkbook.
- Calculate your grade point average.
- Distribute your household expenses.
- Check your credit card debt.
- Keep track of your grade in this course.
- Keep track of your frequent flyer miles.
- Make sure your boss pays you what you earned.
- Convert dates to days of the week.
- Create coded messages.

14. The use of parentheses to group the items together, or indicate scope, is a common computing technique for avoiding ambiguous situations. Without the parentheses it may not be clear whether all the following values are to be added, or just the first two. Dealing with such ambiguities is a major topic in *Chapter 11: Algorithms*.

2	Count the number of items specified:		
3	17	82	=COUNT(B3,C3,D3)
4			
5	Find the absolute value (number without a sign):		
6		-4	=ABS(D6)
7			
8	Remove all upper case letters:		
9	MicroSoft		=LOWER(C9)
10			
11	Find the name of a month:		
12		4	=MONTHNAME(C12)
13			
14	Find part of a date:		
15		=NOW()	=MONTH(C15)
16		=NOW()	=YEAR(C16)
17	Check if a cell contains text or numbers:		
18	abc		=ISTEXT(C18)
19		123	=ISTEXT(C19)

some functions and . . . results

Figure 9.5
Some Typical Spreadsheet Functions

Exercises

9.10 List five more possible uses for a spreadsheet.

9.11 Use a spreadsheet to calculate values for several functions you already know:

- a. The average of your class grades.
- b. The amount you will earn in 17 hours at \$4.35 per hour.
- c. The distance from your town to New York City (add up the individual segment-distances from a map).
- d. The hypotenuse of a right triangle.

9.12 Find five built-in functions you already know and see how the manual describes them. Check how they actually perform in a spreadsheet.

9.13 Find five built-in functions you do not know and figure out what they do. (Hint: use the Help facilities.)

9.14 Find and use in a spreadsheet several common mathematical

functions: sine, cosine, tangent, square root, log, and factorial. Also look for the constant p (pi). Be sure to check your answers by comparing them to values you know.

9.5

Thinking About the Tool

9.5.1

Analogy to Other Tools

The computer application spreadsheet is really one large analogy to a paper spreadsheet. All of its basic concepts of data representation are based on that earlier tool. A paper spreadsheet may or may not be new to you, but your experience with other computerized tools especially databases provides a foundation for understanding the new tool.

Insertion point: as with a database, insertion is always into a single cell.

Selection of a region: the user of a spreadsheet can select individual characters, cells, or entire regions (e.g., a row or a column) of cells, just as with a database.

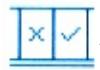
All spreadsheet tools support the basic word processing operations of search, selection, deletion, formatting and so on.

9.5.2

Granularity

Once again granularity provides a key to understanding the tool. The basic unit of operation of word processors and typewriters is the character, but for the outliner it is the topic sentence. The basic unit of granularity for the spreadsheet is the cell. Generally, the operand of any operation is a single cell. You can add two cells, but not parts of a cell. You can work at a lower level by selecting the cell and then performing operations on the individual characters within the cell. A spreadsheet's capabilities at the character level are minimal certainly less than most word processors. For example, an entire cell must normally share a single style or font. You can select a region of cells and manipulate all the cells of the region

together. But unless explicitly stated otherwise, each operation refers to a cell. Think in terms of cells. One of the most common errors made by spreadsheet beginners is that they forget they are working with cells. They begin editing one cell, and attempt to select a second cell without first indicating they are done with the first. Most spreadsheet tools require either a RETURN or explicit selection of the *cancel* or *accept* buttons something like:



before working with another cell. This indicates “I am done working at the character level and am returning to the cell level. Failure to provide this information leaves the cell incomplete, typically resulting in an extra reference in the cell definition. The following miscalculation of a waiter’s tip resulted from an attempt to select cell D6 after completing the definition of D5 without indicating that the first definition was complete.

3	cost of the food	10.13
4	standard tipping rate	15
5	amount of tip	=D3*D4/100+D6

9.5.3

Spreadsheets Versus Databases

The physical appearance and overlap of capabilities of many spreadsheet and database applications can be confusing. Although there are no absolute rules that specify the exact uses of each, there are several rules of thumb. In general,

Use a database if:

- you need a relation with a large number of elements,
- you need to view the data from several perspectives,
- you often need to select a small subset of the whole (especially if you need, say, the largest three values),
- you need to form complex queries, or
- you need to view the data in different orders.

Use a spreadsheet if:

- the problem has several fields or values that can be calculated from other values,
- many fields of different types are interrelated,
- you need “trial-and-error” methods, or
- the global structure is not uniform.

Exercises

9.15 Describe two advantages of a spreadsheet over a database and two advantages of a database over a spreadsheet.

9.16 Give two uses for which a spreadsheet is a better tool, two for which a database is a better tool, and two for which either tool would be appropriate. Explain your answers.

9.6 Summary

A relation in which one value (field) uniquely determines another value is called a *function*. Those functions in which one value can be computed from the other are of particular interest for computer users. The original data is known as the independent variables and the derived data as the dependent variables. Specifying value functionally rather than explicitly makes it easier to extend or modify collections of data.

Spreadsheets are tools designed to take advantage of functions. Any cell of a spreadsheet can be defined functionally in terms of other cells. The tool performs all the needed calculations leaving only the design for the user.

Important Ideas

function spreadsheet
 independent
 variable

matrix cell dependent
 variable

argument

10

Functions and Relations as Problem-Solving Tools

From principles is derived probability, but truth or certainty is obtained only from facts.

NATHANIEL HAWTHORNE

Pointers

*Lab Unit 8: Constructing
Manual: Algorithms Using a
Spreadsheet*

10.0

Overview

A function, as described in Chapter 9, is a special case of relation. Functions also are an essential modeling tool for several classes of problem. Selecting the correct estimate as an approximation of an actual function enables users to evaluate the correctness of their answer quickly. Reciprocally, spreadsheets are excellent tools for developing such rules of thumb.

10.1

Rules of Thumb as Relations

Everyone knows some “*rules of thumb*” guidelines for getting an answer quickly. Usually, such a rule yields only an approximation of the answer. But, for many situations, an approximation is good enough. Most people do not create their own rules of thumb. The rules are handed to us: we read them in a book or hear them from an instructor. The ability to use these rules effectively varies from person to person. Fortunately, a symbiotic relationship between rules of thumb and computing increases the value of each as a

problem-solving tool.

Rules of thumb can help assure that a spreadsheet (or other tool) produces accurate results.

A spreadsheet is useful for exploring possible rules of thumb for future use.

10.1.1

Approximate Answers as a Problem-Solving Method

First, let's understand the value of approximations. The spreadsheet segment in Figure 10.1 includes an exact measure, an *approximation*, and a calculation using

Box 10.1 Significant Figures

Most people actually err more often by calculating values to a far greater precision than their input data justifies. This is the problem of *significant figures* or significant digits that seems to haunt many first-year chemistry or physics students. Much of the problem with significant figures follows from a discrepancy between the actual accuracy of most measuring devices and the ability of a computer or pocket calculator to produce a very accurate looking result. A measurement made with a crude device cannot yield an accurate number no matter how accurate the computational tool. For example, suppose you used a ruler to compare inches to centimeters. You simply measure one inch against the centimeter scale. Few rulers would provide any finer measure than tenths of a centimeter. So you would get the answer: 2.5 centimeters. If you use that value to calculate the number of kilometers in a mile you get:

$$2.5 \times 12 \times 5280 / 100000 = 1.584 \text{ kilometers}$$

an answer which apparently has four significant digits in spite of the fact that the actual answer is greater than 1.6. That is, the last two digits in the derived answer were at best meaningless and at worst very misleading. A good rule of thumb for calculations is:

An answer should have no more precision than the least precise initial input.

In the above example, you should round off the answer to the nearest two-digit answer: 1.6. It comes as a surprise to many people that the rounded-off answer may actually be closer to the “correct” value than the carefully calculated answer.

	feet per mile	miles	total distance	"error"
exact	5,280	98	517,440 feet	0.0%
approximate	5,000	100	500,000 feet	3.4%

Figure 10.1
Exact versus Approximate Distances

each. Suppose you wish to know how many feet are in 98 miles. (Why? Perhaps you are considering installing a private telephone line to your ranch in Montana and need to purchase wire by the foot, but the map gives the distance in miles.) Since a mile is 5,280 feet, you could calculate:

$$\text{feet} = 98 \text{ miles} * 5280 \text{ feet/mile} = 517,440 \text{ feet.}$$

Alternatively, you can approximate the answer as 500,000 feet using the approximate values 5000 feet and 100 miles. That calculation is considerably easier and is “off” by only 3.4 miles (or about 3.4%). This is certainly accurate enough for most purposes. If you were purchasing the wire, you might want 17,000 feet extra in case of damage. If you were getting ready to drive the route, would it make any difference to you if it were 100 rather than 98 miles? Would the trip take much longer? Would you use more gas? The estimated answer is “close enough” for most common purposes. It is far easier to multiply by 5000 than by 5280 (at least for humans), which saves time and frustration. The first two “rules of thumb for rules of thumb” are:

1. Simplify the problem: round all numbers to easy-to-use values. Translate the problem to one using operations which you know well such as multiplying by 10 or 2; avoid multidigit calculations, such as dividing by a number larger than two.
2. Balance the estimates: if you estimate low one time, estimate high the next (5000 was low, 100 was high).

Example 10.1 Metric Conversion. As you know, there are two measuring systems in common use in the world:

English used in the U.S., and

Metric used virtually everywhere else (including England).

Few people are really comfortable with both systems. In particular, most people are not very proficient at converting values from one system to the other. But much of their problem actually comes from trying to get more

1. Actually, the U.S. system differs from the British Imperial system. But the latter is no longer used in England. The name *English* now applies to the U.S. system.

precision than they really need. All they really need in most situations is a feeling for approximately how large a unit is. Consider the rule of thumb:

*To convert from metric to English,
multiply by 1.1.*

For example, multiply the number of liters by 1.1 to get the number of quarts. The actual ratio is really 1.06. If you fill your car with 40 liters of gas, you could calculate the number of gallons by multiplying the number of liters by 0.265 (to get 10.6 gallons). But it's far easier to convert the quantity to 44 quarts (40×1.1 quarts/liter) and then convert that to 11 gallons and you will only be off by less than half a gallon. Since few numbers are easier to multiply by than 1.1, this seems like a nice method.

At first glance this rule of thumb seems outrageous if applied to weights or distances. After all, a meter is 39.37 inches. Multiplying a meter by 1.1 gives you a good approximation of neither feet nor inches. But it does give you yards:

meters	yards	inches
1	1.1	39.60

OK, so it's off by 0.23 inch. But it is very close. An even easier approximation might be "a bit larger than the number of yards." Track and field athletes probably already know this conversion because 400 meters is almost exactly 440 yards; 800 meters » 880 yards; 1600 meters » one mile. Knowing that 800 meters is a half mile helps a lot in visualizing a kilometer (1000 meters). The following table summarizes this rule of thumb:²

Metric	is approximately
meter	1.1 yard or 3.3 feet
kilogram	2.2 pounds
liter	1.1 quart or 1.1/4 gallons

metric ton 1.1 ton or 2200 lbs.
nautical mile 1.1 mile

10.1.2

Spreadsheets as Tools for Developing Rules of Thumb

A spreadsheet can be used for developing, testing, or understanding a rule of thumb. One problem many people experience goes something like this: they once

2. Obviously a nautical mile is not a metric measure. It is included here because it does fit the rule of thumb, making the rule even more useful. On the other hand, no temperature conversions are shown. Exercise 10.3 asks you to derive a similar rule for temperature conversion.

knew a mathematical formulathey even knew it well enough to pass a test. They have not used it since. Now they remember it only vaguely. To use the formula they will need to either retrieve the exact value or generate a good approximation.

Example 10.2 Temperature Conversion. Your friend in New Zealand just wrote to you that it is a beautiful summer day and the outside temperature is 31 degrees. Wait! That sounds cold. Oh, they use the metric system there, so that must be the Celsius temperature. But is 31° hot, or warm, or just “not too cold”? One obvious technique would be to check the metric rule of thumb. Does that work? Alternatively, you could apply the temperature conversion formula:

$$\text{Fahrenheit} = \frac{9}{5} \times \text{Centigrade} + 32 \quad (\text{EQ 10.1})$$

or was that

$$\text{Fahrenheit} = \frac{5}{9} \times \text{Centigrade} + 32? \quad (\text{EQ 10.2})$$

or even

$$\text{Fahrenheit} = \frac{9}{5} \times (\text{Centigrade} + 32)? \quad (\text{EQ 10.3})$$

This correct version of this familiar formula can provide the answerbut which one is correct?

You can use a spreadsheet and a little knowledge to select the correct formulaand generate a good rule of thumb for next time. What facts do you know? Water boils at 212°F or 100°C and freezes at 32°F or 0°C. (Perhaps you can’t remember if its 212 or 221 or “two hundred and something.” That’s good enough for generating a rule of thumb). So, build a spreadsheet to help. Use the values you do know and try each candidate function and see which comes out closest:3

24	Centigrade		Fahrenheit	
25	freezes	boiling point	=A25	=B25
26	0	100	=5/9*A26+32	=5/9*B26+32
27	0	100	=9/5*A27+32	=9/5*B27+32
28	0	100	=9/58*(A28+32)	=9/58*(B28+32)

(You do not even need to know these numbers exactly to come up with an approximation.) The calculations for the freezing point yield an ambiguous

3. Notice the use of functions in cells C25 and D25 to assure that the labels are identical in both sections.

result: the first two formulae seem to work. Including the boiling point provides a definitive answer: only the second formula gives correct

	Centigrade	Fahrenheit	
	freezes	boiling point	freezes
24	0	100	32
25	0	100	32
26	0	100	5
27	0	100	212
28	0	100	20

values for both anchor points. Therefore use that formula to calculate:

$$9/5 \times 31^\circ\text{C} + 32 = 88^\circ\text{F}.$$

But the original problem does not even require that much detail. You could make a rule of thumb based on the actual formula. Clearly $5/9$ is about $1/2$. That is, each Fahrenheit degree is about half the size of a Centigrade degree. Thirty-two is about 30. So roughly: 31°C is about $30 + 30 * 2$ or about 90°F . It must be fairly hot there right now and that is, after all, what you wanted to know. You could use the spreadsheet to check several values as a test of the accuracy of the rule.

Notice the two-way street in the above example. The conversion represents a search for a rule of thumb. In reality the technique can be used in any situation requiring relationships about which you have confusion. The general methodology is:

*Generate an approximation for a relationship.
Test the proposed approximation.
If the resulting value makes sense
then the relationship may be correct,
otherwise you have made an error.*

That is, the approximation serves to check your understanding of the situation. More important, if the approximation yields an incorrect answer, your understanding is likely flawed and you need

to rethink your answers.

Exercises

10.1 The rules of thumb for converting metric weights and volumes to their English equivalents are almost identical to the one for distances. Find them! Use a spreadsheet to compare the rule and the exact. To get you started, here are some (not necessarily the most useful, but good enough if you have a spreadsheet) of the exact conversions: 1 pound = 453.592 grams, 1 gallon = 3.785 liters.

(table continued on next page)

(table continued from previous page)

- 10.2 Use a spreadsheet to check the accuracy of the Fahrenheit conversion rule of thumb. Use the rule of thumb to check your spreadsheet.
- 10.3 Example 10.1 derived a rule that could be called the "multiply by 1.1" rule for converting metric to English measures. The rule apparently did not apply to temperature conversion. Derive a rule for temperature conversion (see Example 10.2) that uses the 1.1 factor. (Hint: it is not quite as simple as the others.)
- 10.4 A common rule of thumb for trigonometry is "for small angles, the sine of the angle is approximately equal to the angle. Test this rule for several angles. What must units of measure be for the angle?"
- 10.5 For each of the approximate conversions in this section, find the exact values. Find the English equivalent of each of 100 units of the similar metric unit (e.g., how many feet is 100 meters). What is the difference between the result found using the approximation and the result found using the actual measure? (Of course you may use a spreadsheet.)
- 10.6 Although most word processors include a tool for counting the number of words in a document, this tool does not help the reader of a completed (paper) document. Devise a technique for estimating the number of words in a book. Use that technique to estimate the number of words in this text.
- 10.7 Compare the following approximations to their actual values. For each determine the difference between the approximation and the exact value. You may find the difference for a single unit or for a larger conglomerate.

A month is 30 days (find error in a month or a year).

There are 150 hours in a week.

There are 10,000 seconds in a day.

Annual salary (in thousands of dollars) is twice hourly rate of pay.

10.1.3

Rules of Thumb for Verification

Rules of thumb are also powerful tools for checking the results of your computation. We all like to think we can rely on the answers we get from our computer. Alas, this is just too optimistic not because the computer may have made an error but because the human user may have incorrectly specified the problem.

Verification in Everyday Life.

One of the most important uses for approximations is as an aid in recognizing plausible answers quickly, or “on the fly.” The user needs to have confidence in the answer produced by the tool. Rules of thumb and other approximation techniques can aid this confidence. This is certainly important in the computer world, but there are also many uses of rules of thumb for verification in the noncomputer world. For example, suppose you go to the grocery store and purchase three quarts of milk, a dozen eggs, and a loaf of

bread. The cashier says “Thank you. That will be \$32.17.” Clearly something is wrong here (either that or groceries in your town are very expensive). If you detect a problem, you can correct it and pay the right amount. If you do not notice the problem, you lose about \$27 not a very nice situation. A good approximation technique that will help highlight such errors need only be accurate enough to separate the plausible from the implausible answers. Where did the estimate of the error (\$27) come from? One approximation technique is to estimate the cost as:

Groceries cost approximately \$1 per item.

The customer in the example purchased five items and was charged about \$32\$27 too much. The rule may even provide a hint about what is wrong. There was more milk than anything else in the purchase. If each milk were overcharged \$9, that would make \$27. If the decimal point were in the wrong position, the charge would be about \$10 instead of about \$1, that’s exactly the amount of the error.

Many students already employ rules of thumb. Most readers would recognize that the above purchase price was wrong if they had been in the situation. Their rule of thumb might be as simple as:

“I don’t have that much money with me, but I was sure I had enough for the groceries. Let me look to see what is wrong.”

Applying Rules of Thumb to Spreadsheets.

Such rules are especially useful for computing. It is very easy to make a typing error when entering numeric data, or to get the expression for a formula slightly wrong. A spreadsheet will dutifully produce the correct answer to the wrong calculation. As far as you are concerned, it is an incorrect answer. It is up to you to recognize a wrong answer. Rules of thumb will help you recognize, at least, the more obnoxious errors. The next two rules of thumb for rules of thumb apply to verification:

3. Select easy-to-calculate data.

Ideally, the correct answer or dependent variable should be approximately the same as one obtained using a rule of thumb. So the values used for independent variables should not be too different than the actual values, but they should be ones that provide for simpler arithmetic. For example, when calculating a waiter's tip, use \$10 as the amount of your dinner, not \$13.97. Calculating the expected answer in your head makes many errors obvious. Ten percent of ten dollars is one dollar. Fifteen percent should be more than that but not twice as muchsomething like \$1.50. Anything that isn't close to that is a potential problem. The source of the error may not be immediately obvious, but recognizing the existence of an error is at least half the battle.

4. Estimate the expected answer first.

Whenever you specify a function in a spreadsheet, estimate the answer you expect. Use one or both of the following techniques:

round off the numbers and approximate in your head.

use easy numbers as data to start with. Use the actual values when you are confident of your function.

But perform the estimation first. With no expected answer, no errors will be obvious. For example, suppose you wish to calculate the area in square centimeters of a circle with a one-inch radius. You know the formula:

$$\text{area} = \pi r^2$$

You could just plug the values directly into a spreadsheet:⁴

Area of a circle		
radius	2.54	
area	=3.141592653*J2^2	=PI()*J2^2

but will you know if it is right? A reasonable approximation is that the answer should be a little less than $3 \times 3 \times 3$ (round both values, one high and one low) or 27. Compare the approximate answer with the derived answer (20.3). If you performed this calculation with no expectation and found an answer of 40, it might seem completely reasonable. The use of rules of thumb and recognition of errors is a major topic in later chapters.

10.2

Calculations with Multiple Answers

Many problems have more than one answer. Multiple answers may represent multiple functions applied to a single set of data or a single function applied to multiple sets of independent data.

10.2.1

Complex Relations

Several results may be derived from a single data set. For example, some problems have two alternative correct answers (e.g., the roots of a quadratic equation). Many more problems have two distinct values that are of interest. For example, the tipping example in

Figure 9.3 could easily be expanded to include a tax calculation, as in Table 10.1 on page 176. Even though the definition of functions implies that it should have a single result, they can be powerful tools for such problems. Any spreadsheet can contain any number of calculations as if each were the only value of interest. The table contains the functions

$$\begin{aligned} &\text{amount of tip (basic food cost, tip rate)} \\ &\text{Total tax (basic food cost, tax rate)} \end{aligned}$$

The two functions share the independent variable, *basic food cost*. Each has its

4. Notice the use of the built-in function *pieasier* than remembering the exact value!

TABLE 10.1 Spreadsheets and Tables

Cost of the food =	\$10.13
Standard tipping =	15%
rate	
Tax rate	= 7.5%
Amount of tip	= cost of the food × standard tipping rate/100
Total tax	= cost of the food × tax rate/100

own result or dependent variable. This actually suggests a more complex relation with more fields:

expenses (basic food cost, tip rate, tax rate, tip, tax)

In this case both *tax* and *tip* are dependent variables. The computations for both results can be included in the same spreadsheet. Notice that calculating the total cost of the entire meal including the tax and tip may require extra care because the grand total depends on the other calculations. The additional function:

total cost = cost of food + amount of tip + total tax

will calculate *total cost*, but both the *tip* and the *tax* must be known before calculating the *total cost*. Attempting to calculate *total cost* before calculating the other two values will produce a meaningless answer. The next chapter introduces the concept of *algorithm* as a technique for avoiding this and related problems.

10.2.2

Spreadsheets and Tables

Another common use of spreadsheets requires the calculation and display of several elements of a relation “side by side” for comparison.

Example 10.3 Beer and wine have a much lower alcohol content than do the so-called hard liquors. For this reason, some drinkers

prefer these beverages for parties and other situations in which they may have several drinks. How much better are beer and wine than hard liquors for reducing total alcohol consumption?

The problem can be restated more precisely:

How much less alcohol is in a glass of beer or wine than in a mixed drink?

For any drink, the amount of alcohol it contains can be found by the function:

$$\text{total alcohol} = \text{percent alcohol} \times \text{volume of liquid}$$

This one formula can be used multiple times in a spreadsheet to find the alcohol content of each of several drinks. The only independent variables are:

the size of the drink and the percent alcohol (in mixed drinks, the percent of alcohol refers to the volume of the alcoholic beverage only not to the mixer). To find out how much better beer is, create a small spreadsheet:^{5,6}

beverage	% alcohol	size of drink in ounces	total alcohol content (ounces)
beer	5	12	0.6
wine	12	5	0.6
rum	40	1.5	0.6

Figure 10.2
Testing an Assumption

Surprise! The total alcohol content is the same. A beer has just as much alcohol as a mixed drink! Does this mean your assumptions are incorrect or that the spreadsheet representation is correct? Unlike the approximation examples, this set of answers does not indicate either errors or correctness. The point was to create the table of information.

Such tables are very common. In effect a table is the listing of the elements of a function or relation. Alternatively put, the table applies the same function to multiple sets of independent variables. Each row represents one element, the items in that row, the fields of the element. Other examples of table-structured problems include lists of:

angles and their sines (as in a trigonometry book).

purchase price and applicable taxes (you might find this at retail stores).

dates and the corresponding days of the week.

dates and the expenses for that week.

Such tables may actually represent one of the most common uses of spreadsheets. A related example, “what if” tables, is described in Section 12.4.

Exercises

10.8 Create spreadsheets depicting the following multiple-values functions:

The maximum, minimum, and average of three values.

The predecessor and successor of a letter of the alphabet.

The square, cube, and square root of a number.

(table continued on next page)

5. The data on this spreadsheet represent a typical American beer, a bottle of California Chardonnay, and an 80 proof bottle of bourbon. Percentages do vary slightly.

6. Notice the repetitious nature of this calculation each calculation is essentially the same as the others. *Chapter 15: Iteration: Replicated Structures* discusses this command subclass of problems.

(table continued from previous page)

10.9 Create spreadsheets depicting the following tables:

the cosine of angles between 0 and 90° .

a standard multiplication or addition table.

the successor of each letter of the alphabet.

10.3

Thinking About Functions

10.3.1

Dealing with Representation

Many seemingly innocuous errors can cause major problems in any computation. Be wary of your notation. Recall the function for calculating a waiter's tip:

cost of the food	\$10.13
standard tipping	15
percentage	
amount of tip	$\text{cost of the food} \times \frac{\text{standard tipping rate}}{100}$

A percentage is actually an alternative representation of a decimal fraction. Each percent value represents the number as hundredths. The use of a percentage in most calculations requires that you use the percentage divided by 100 rather than the percentage expressed as an integer (e.g., 0.15 rather than 15). Although none of this is new or surprising, many errors are caused by sloppy writing of functional values.⁷ This is also a good place to use a rule of thumb. A miscalculated tip, in which the cost was multiplied by 15 would result in an answer of over \$150. Fortunately, a very simple rule of thumb suggests that the tip should not be larger than the cost of the dinner.

10.3.2

Dealing with Generality

The spreadsheet for calculating a tip (Table 10.1) includes a cell containing a fixed value—the standard tipping rate: 15%. The function definition refers to that cell:

$$\text{amount of tip} = \text{cost of the food} \times \text{standard tipping rate}/100$$

rather than including the number explicitly:

$$\text{amount of tip} = \text{cost of the food} \times 15/100$$

Inclusion of an explicit value within a function definition called a *literal* value is always legal, but it reduces the flexibility of the definition. Although the standard tipping rate is relatively constant, it may change over time (tipping rate

7. As an alternative to dividing by 100, most spreadsheets allow you to *format* the cell as a percent: the user enters the value as 0.15, but it shows up as 15%.

is, after all, controlled by custom). If, a year from now, society concludes that food servers are underpaid (and they probably are), the standard tipping rate could change to, say, 20%. A *hard coded* literal (e.g., the 15%) in a definition causes a problem when changes occur. A change in tipping custom will require finding and changing all tip rates that are hard coded as literal values. Finding such values requires examination of the function in each cell. Instead, the tip rate can be coded in its own cell, as in Figure 10.3, and each function needing the value can refer to that cell. Then when changes occur, only that one cell will need to be changed; the functions themselves can remain fixed. In general, if there is any possibility of a value changing in the future, enter it in a separate cell. Function definitions should contain very few literals.

10.4

Techniques for Describing Functions Using Spreadsheets

A spreadsheet is an excellent tool for describing functions between data items, just as a word processor is an excellent tool for recording textual data. But like the word processor, it does not do the job by itself; it requires a human operator. As with database tools, there are good ways and bad ways to organize material in spreadsheets. Actually, the rules for each are quite similar. Following are several rules to help you start creating good and productive spreadsheets.

Distinguish Initial from Derived Data.

The reader of the data will want to know which of your data items came from external sources and which you (or your computer) derived. A printed spreadsheet does not indicate to the reader which is which. If the reader does not think the answers are correct, there is no way to know whether the complaint should be directed at you or at your information sources. The specific method used to distinguish between these two sets is not as important as the clarity of the result. One approach is simply to divide the data into two

sections in independent and dependent values. In Figure 10.3, these were shown as given and derived data respectively. Notice also that the

Given data		
	cost of the food	\$10.13
	standard tipping rate	15%
Derived data		
	amount of tip	\$1.52

Figure 10.3
Proper Use of “Constant” Values

spreadsheet is *formatted*, with the two categories divided by a pair of lines. Each of the sections is labeled to indicate whether it is input directly or derived. The titles “given” and “derived” are a bit general. In many situations, alternatives will suggest themselves (initial values, input, “please enter”, and so on, versus results or “the answer is”).

Like a database, a spreadsheet is normally used interactively. The general format is set up by the designer, but a different person may enter the data. That person will want to know which fields are to be entered and which the spreadsheet will calculate. The distinction may be made clear through instructions to the user, as in this version of the roommates’ rent calculator:

enter monthly rent:	\$ <input type="text" value="900"/>
enter the number of roommates:	<input type="text" value="3"/>
then, each person's rent is:	\$ <input type="text" value="300"/>

The instructions clearly indicate which items the user should enter. In addition, the borders around the cells make it clear where the values go. In this example, the cell grid is turned off in the display for added clarity.⁸

Format for Clarity.

Information should be easy to find. Have you ever looked all over the page to find where to put your signature? Or missed the “fine print” on a contract? Organize the document to help the reader or user find the important material. In particular make the “bottom line” or final answer easy to find.

Label all data cells with a name or phrase that makes the meaning of the contents of the cell clear. Use an adjacent or nearby cell for the label.

Place input in one area (usually the top) and derived data in another.

Place important items (e.g., the final answer) in positions of prominence often offsetting these values one column to the right will serve this purpose.

Highlight important items. Use bold fonts or distinguish borders of important cells. Color, shading, or size may also be possible.

Do not clutter the document with too many fonts and styles in ways that distract from the important items.

8. See Section 8.2.1 of the lab manual, or use the Help facility to look up menu items such as *Format*, *View*, and *Display*.

Use blank space. Don't jam all the values into the smallest possible space. Spread them out for easier reading. Place extra space between logical segments of the spreadsheet so that it is easy to see the individual parts.

Unfortunately, the previous rule must be balanced against screen size. It is useful if all related items visible are on the screen simultaneously. Keep items near each other so that the user does not need to scroll over many pages.

Format the individual cells wisely. Make sure every cell is wide enough to display the value it holds. Make sure the format of the cell is appropriate for the data. If the cell contains a monetary figure, show the dollar sign and do not allow 5 digits to the right of the decimal point.

Remember that People Forget.

These guidelines apply even if the spreadsheet is for your own personal use (that is, you are the user as well as the designer of the spreadsheet). People forget and that includes both you and me. This means that in any complicated spreadsheet you are likely to forget the definition of some value or how you calculated it. (This will become even more important in the next chapter.)

Exercises

10.10 How would you verify the assumptions used for the alcohol content problem (Figure 10.2)?

10.11 A typical bank credit card charges about 19% per year. How much is that per month? If you charge \$1234, how much will you owe in interest the first month? How much is that per year?

10.12 Many people have trouble budgeting or even keeping track of their money. Some “money professionals” recommend the

following technique for getting started. Write down everything you spend for two months. At the end of that time, go through and classify each item as essential, optional, or “needed, but could be reduced.” Add up the amounts in each category. Create a spreadsheet version of this and record every item faithfully for two weeks.

10.13 List five more possible uses for a spreadsheet beyond those that you thought of when first asked to do so in Exercise 9.10.

10.5 Summary

A function is one of the fundamental problem-solving tools of a computer scientist. A functional calculation allows users to specify the relationship and allow a computer to perform the hard work. Functions not only provide direct answers to calculations, but they also provide a means of checking the accuracy of other results.

Important Ideas

approximation testing error

table multi-valued format
function

significant rule of
figure thumb

11 Algorithms

I always begin at the left with the opening word of the sentence and read toward the right and I recommend this method.

JAMES THURBER

Pointers

Gateway Arithmetic

Lab: expressions

*Lab Unit 8: Constructing
Manual: Algorithms Using a
Spreadsheet*

11.0 Overview

The simple relations and functions of Chapters 8 through 10 are powerful tools for constructing organized collections of data. However, many problems require even more powerful tools. Although the built-in functions and operators are the most commonly needed tools, seldom does any one tool provide exactly the needed result. Fortunately, it is possible to combine existing tools in new combinations tailored to a specific problem. A systematic and organized approach to creating such new combination tools is based on *algorithms*.

11.1 The Definition of Algorithm

The formal definition of *algorithm* is:

A finite ordered sequence of unambiguous steps that leads to a

solution for a problem.

This is often shortened to just:

A sequence of steps that leads to a solution to a problem.

Some examples of algorithm include:

Directions to a strange location:

*Take I-390 south.
Get off at exit 8.
Take Route 20A west.
Turn right at Main Street.
Turn left at Park Street.
Turn right at College Circle.
Stop at the flag pole.*

The recipe for Tripe a la mode de Caen: 1

*Trim and wash 3 lbs of fresh tripe.
Cut tripe into 1 inch squares.
Blanch a split calf's foot.
Peel and slice 2 lbs of onions.
Dice 1/4 lb beef suet.
Place onions, tripe, and suet in casserole in layers.
Top with the calf's foot, onion (stuck with 3 cloves) and a bay leaf.
Add 1/4 cup brandy and water.
Bring to boil.
Seal casserole with pastry dough.
Bake for 12 hours.*

Box 11.1 al-Khowârizmî: A Very Early Computer Scientist

The word *algorithm* is much older than any computer. It is named after a ninth-century Persian mathematician: Abu Ja'far Mohammed ibn Mûsa al-Khowârizmî, who described the process of performing arithmetic computations using Arabic numerals a major improvement over procedures that used their *Roman* counterparts. As important as his work was at the time, no one could possibly have imagined that it could form the basis of an entire discipline 1100 years later.

1. Adapted from *Joy of Cooking* by I.S. Rombauer and M.R. Becker (New York: New American Library, 1964).

*The directions for assembling a new computer:*2

Place computer on flat work surface.

Plug computer in.

Connect monitor:

Place monitor on top of computer.

Attach power cord to monitor.

Attach video cable to monitor and computer.

Connect mouse.

Connect keyboard.

Turn computer on.

In fact you have seen many algorithms in this text, such as the slightly more complicated algorithm (copied from Section 7.3.1) describing the use of a paint tool for drawing a diagram of a room:

First draw the room as a rectangle.

For each window:

Erase the part of the rectangle where the window goes.

Draw a simple rectangle representing a window.

For the door:

Erase the part of the rectangle where the door goes.

Add the individual parts of the door:

To make the arc

draw a circle and

then erase the parts not needed.

Draw a line from the wall to the arc.

In general, any set of directions in this text displayed in this manner is an algorithm or at least very close to an algorithm. As you will see, the details of the definition will disqualify some apparently well-written instructions.

Algorithms provide the essential mechanism for specifying instructions to a computer. In fact, a fundamental thesis of computer science is that anything that can be computed by any method can be

computed using an algorithm (see Box 11.2). Conversely, this simple class of instructions is sufficient for calculating any computable function. By combining appropriate sequences of basic functions, we can perform any computation. For example, the description of the function for calculating the cost of a restaurant meal (in Section 10.2.1) suggested that calculating the total cost, including a tip, has an extra complexity: the order of the steps is essential because the calculation of the total cost depends on the calculation of the tip. Therefore, the tip must be calculated before the total. Table 11.1 on page 186 shows the essential relations. What is needed is a formal statement of the algorithm for deriving the results:

2. Adapted from *Getting Started with Your Macintosh Centris*, Apple Computer, Inc. 1992.

1. Find (and display) the cost of the food.
2. Find (and display) the tipping rate.
3. Calculate the tip (tipping rate times food cost).
4. Calculate the total (tip plus food cost).

Notice that the algorithm includes determining independent variables: the cost of the food and the tipping rate (perhaps they are read from the menu, known from a previous visit, or quoted by the server).

Box 11.2 Two Men, Way Ahead of Their Time

Alonzo Church (an American) and Alan Turing (an Englishman), working independently, both developed the conjecture that “any effectively computable function could be computed on a very simple computer” (called a *Turing machine*). In simpler terms, this means that all computer languages are equally powerful, or alternatively that “any algorithm that can be described in English, can be computed on a computer.” At the time of their work, no electronic computer hardware had been built. To this date, no Turing machine has ever been built. Yet, it is now universally accepted that they were correct. This means that they both figured out the basic principles describing what could and could not be computed without ever seeing a computer.

In addition to almost single-handedly defining the future field of computer science, Turing was instrumental in breaking Axis military codes during World War II. It is a sad commentary on twentieth-century prejudices that in spite of the importance of his work, he was persecuted for his sexual preferences, and eventually committed suicide.

11.1.1

Exploring the Definition of Algorithm

Although the definition of algorithm may seem unwieldy, each part

of the definition is important for assuring that an algorithm will be a useful tool.

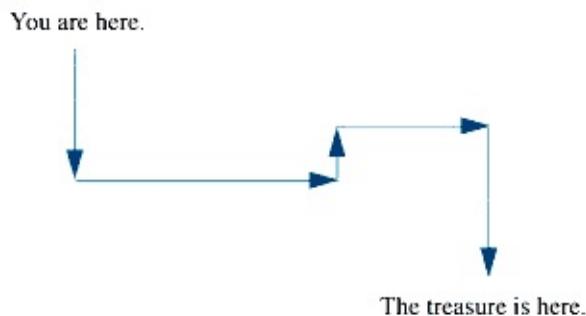
TABLE 11.1 A Restaurant Meal
with Tip

Cost of the food:	\$10.13
Standard tipping rate:	15%
Amount of tip:	\$1.52
Total cost:	\$11.65

Steps.

Specifying precise individual steps is essential. General guidelines are not sufficient. Although a map may be useful in getting to a destination, most maps are not algorithms. Why? Because they do not list the steps for getting to that goal. A map is actually just a set of facts that can be used as building blocks for an algorithm. Some specialized maps include a complete algorithm. The directions to “a strange place” (in Section 11.1) could be written on a map.

Alternatively, a map may include arrows showing the intended path. A map with arrows does represent an algorithm because the arrows explicitly indicate each step and its relation to the other steps.



A step-by-step algorithm helps ensure that the follower does exactly as the designer intended. A map follower may not take the route that seems obvious to the designer. In addition, providing specific steps makes complex processes clearer. Whenever instructions allow the possibility of doing two things at once, as in

Add the eggs and continue stirring,

ambiguity results. If the algorithm containing this instruction does not also explicitly say when to stop stirring, some followers may stop shortly after adding the eggs; others may stir until the entire task is complete. Simultaneous actions also create an ambiguity by suggesting actions that some may find impossible: only experienced cooks can stir with one hand and break eggs with the other.

Ordered Sequence.

The most important words in the definition of *algorithm* are “sequence” or “ordered sequence.” A meal-cost-with-tip algorithm with step (3) before step (2) will not work. There is no way to calculate the total cost without previously finding both items that must be added together. The tip must be calculated before adding the total. With the order reversed, no value for the tip would be available to use in calculating the total. Errors in the other example algorithms would create analogous problems (imagine sealing the casserole before adding the ingredients!). Consider two short sets of directions:

Go north one block.

Turn right.

Go three blocks.

Turn left.

Go one block.

and

Go north one block.

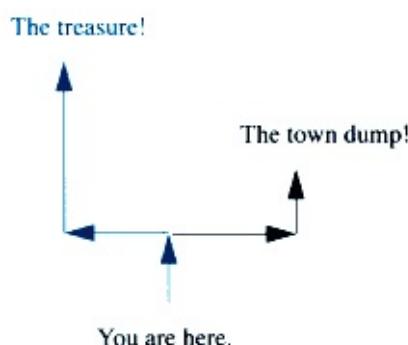
Turn left.

Go one block.

Turn right.

Go three blocks.

The two sets of directions have identical steps but differ slightly in the order. Unfortunately, even that small variation leads to two distinct locations:



Unambiguous.

A map provides many ways for getting from point A to point B. It does not specify which of the many roads the user is to follow. It also provides information about getting to other destinations. The map reader must bring additional knowledge to the process: heavy lines represent major roads, some rivers may look like roads, the construction project on one route would cause a delay, and so on. The map contains only part of the information needed. The reader must generate a single algorithm from all this information. The definition requires that the algorithm be unambiguous, that it

specify all of the needed information. Thus, “Drive to the store,” is probably ambiguous because it does not specify the route. Fortunately, few (if any) computer commands are ambiguous.

Solution.

An algorithm must lead to the solution or answer to a problem. It is not a rule of thumb. It does not find an approximate answer. It doesn’t sometimes work, and sometimes not. It must guarantee that following the steps will lead to a solution. This guarantee increases the usefulness of an algorithm as a tool for computing. Knowing that the steps do lead to a solution, the user can confidently ask a computer to follow the algorithm. But the requirement also means that the author of the algorithm must consider all possibilities.

Finite.

This final requirement sounds silly at first. How could anyone write an infinite series of steps? Actually, this word embodies two separate requirements. First, an algorithm must be describable in a finite number of lines. Second, the algorithm must actually reach the conclusion in a finite amount of time. The “finite” requirement helps explain the examples of noncomputable function described in Box 9.1. Any proposed algorithm for calculating the exact decimal representation of one-third (0.333333 . . .) is doomed to violate this guideline because no matter how many 3s you write down, there is always a more accurate answer to be found by adding one more “3”. Section 12.3 includes other examples of proposed algorithms that fail to meet this requirement.

11.1.2

Algorithms, Computer Instructions, and Computer Science

Algorithms are the primary means by which users give instructions to computers. In a very real sense every instruction made by a user is a short (one-step) algorithm. Many actions require more complex algorithms. For example, one algorithm for creating the font used in this text for product names (e.g., *Microsoft Word*) is:

Make the font Helvetica.

Make the style italics.

The field of computer science has actually been defined as the study of algorithms. Understanding the relationship between algorithms and computers reduces confusion that some people have when using computer tools. Algorithms will be your primary tool as you create more complicated instructions for computers.

11.2

Arithmetic Expressions as Algorithms

Tasks requiring algorithms occur in every context—many so familiar

that they often go unnoticed. A common, but important, class of algorithms is the *arithmetic expression*, such as (EQ 11.1). The order of the steps is essential for understanding or interpreting arithmetic expressions. The expression

$$3 \times 4 + 5 \quad (\text{EQ 11.1})$$

specifies two operations, which, in turn, allow two possible interpretations, or orders, of execution:

(a) *multiply* $3 \times 4 = 12$
then add $12 + 5 = 17$

or

(b) *add* $4 + 5 = 9$
then multiply $3 \times 9 = 27$

Whoever or whatever interprets $3 \times 4 + 5$ must select an order for the two

operations. The writer of the expression must make the intended order clear. The rationale for picking one interpretation over the other may be well reasoned; perhaps it was based on the *precedence* rules of linear algebra (which state that algebraically, multiplication has higher precedence than addition and therefore expressions are disambiguated by performing multiplication first). Or it may represent a simple preference, such as, "This is the one I intended, so do it that way." At this point, many (or even most) textbooks would describe the interpretation used by a particular software product. This text does not provide a specific interpretation. In general, you should devote relatively little time to learning or memorizing the interpretation used by any specific software. That interpretation is a relatively arbitrary convention, which may vary from one application to another.³ The time spent memorizing semantic rules will be better spent learning to create and specify unambiguously your own correct algorithms. There are two good alternatives to memorizing the precedence rules that determine the correct interpretation of (EQ 11.1):

1. Avoid the problem by forcing the machine to accept your intended interpretation (Section 11.2.1), and
2. Determine the rules used by your particular computer application by *empirical* methods (Section 12.2.1).

11.2.1

Unambiguous Representations can Force Unambiguous Interpretations

Normally, the writer of an arithmetic expression has an order or at least an interpretation in mind. Perhaps (EQ 11.1) represents a calculation of the cost of some new clothes, for example:

3 pairs of socks at \$4 per pair, and a \$5 shirt

or perhaps it is:

3 outfits, each composed of a \$4 shirt and a \$5 tie.

Either way, it is always possible to write arithmetic expressions that unambiguously represent the intended interpretation. In fact, there are at least two ways to do so.

Use Intermediate Results.

Translating the expression into an algorithm with substeps and intermediate calculations forces the order of evaluation. For example, an algorithm with the explicit individual steps:

Find the individual costs.

Calculate the cost per outfit.

Calculate the total cost.

3. Admittedly, it is likely that every application interprets this simple example in the same way. But they do vary in more complicated examples, for example $a \times b/c$ or a^b^c .

	A	B	C
1	Prices		
2		shirt	\$4.00
3		tie	\$5.00
4	Costs		
5		outfit	=C1+C2
6	Number purchased		3
7			
8		total	=C5*C6

Figure 11.1
Cells Dependent on Other Cells

must represent the second interpretation. The cost of one outfit represents an intermediate value. The equivalent spreadsheet (Figure 11.1) shows explicitly that some cells are dependent on others. This representation clearly indicates (to both the reader and the spreadsheet application) that the total (C8) is derived from (dependent on) cells C5 and C6. Therefore those values must be calculated before the total. Likewise, the outfit price (C5) is calculated from the shirt (C2) and tie (C3) prices, so the individual prices must be known before the outfit price can be calculated. The spreadsheet application will always calculate all independent values before attempting any cells dependent on those values. Thus the algorithm described in the spreadsheet is:

Calculate C2 and C3 (shirt and tie prices).

Calculate C5 (outfit cost).

Calculate C8 (total cost).

exactly as required.

Parenthesized Expressions.

Parentheses can also be used to force a specific interpretation:

$$(3 \times 4) + 5$$

means

*Multiply 3 by 4, and
Add 5 to the product.*

In contrast,

$$3 \times (4 + 5)$$

means

*Add 4 and 5, and
Multiply the sum by 3.*

Each expression is equivalent to an algorithm for computing the value. The interpretation of an expression with parentheses is:

1. *Perform all calculations inside the parentheses first.*
2. *Use that result in later operations.*

Since the use of parentheses forces the order of evaluation, it also specifies an algorithm, in this case the same algorithm specified above using intermediate steps. The alternative algorithm and result could be specified by

$$(3 \times 4) + 5.$$

Scope.

Sometimes algorithm designers need general words for describing the distinctions illustrated by the choice of arithmetic algorithms. We say that the operands of each operation are in its *scope*, or that the *scope* of an operation includes all of the things to which it applies. The above explorations can be restated as an attempt to determine the relative scopes of the two operations. Syntactically, the scope of a binary operation is generally the two operands surrounding the operation.

Nested Parentheses.

The techniques also apply to more complicated algorithms. For example, suppose your instructor informed you that your grade in this class would be based on:

Twice the sum of your homework and lab scores, plus your test score.

One way to calculate that value is to specify the algorithm as individual steps:

*Find the individual scores:
lab
homework, and*

test.

Add the lab and homework scores together.

Double that sum.

Add that result and the test score.

An alternative representation uses a single arithmetic expression with two nested pairs of parentheses:

$$\text{total score} = (2 \times (\text{lab score} + \text{homework score})) + \text{test score} \quad (\text{EQ 11.2})$$

Notice that the parenthesized expressions must be interpreted from the inside out. The rules for evaluating a parenthesized expression say you should first evaluate:

$$(2 \times (\text{lab score} + \text{homework score})). \quad (\text{EQ 11.3})$$

But that expression also contains a parenthesized expression

$$(\text{lab score} + \text{homework score}) \quad (\text{EQ 11.4})$$

which must be evaluated first. Evaluating the inner expression and then the outer yields the algorithm above. Alternatively, we can say that the parentheses indicate the scope of the operations. The scope of the outer pair of parentheses includes the scope of the inner pair.

The Pythagorean theorem for finding the hypotenuse of a right triangle is:

$$\text{hypotenuse} = \sqrt{a^2 + b^2}, \quad (\text{EQ 11.5})$$

where a and b are the lengths of the other two sides. The *vinculum* (Ö) means *take the square root of what follows*. In English, the reference to “what follows” serves to group several items together. Thus, it also acts as a set of parentheses. Converting (EQ 11.5) to a series of steps gives:

- Find a and b (it doesn't say which of these must be found first).*
- Calculate the square of a and the square of b
(again, it doesn't say which is first).*
- Find the sum of the squares.*
- Compute the square root of the whole thing.*

Every (parenthesized) arithmetic expression similarly dictates an algorithm for its evaluation and vice versa.

Exercises

- 11.1 Give an algorithm for getting from your lab room to the cafeteria.
- 11.2 Give directions for opening a spreadsheet on your computer.
Assume the computer is turned off at the beginning of the algorithm.
- 11.3 Suppose you have a bibliographic citation such as Rombauer, I.S. & M.R. Becker. Joy of cooking. New York: New American Library, 1964.

Write an algorithm for using your word processor to format it appropriately:

Rombauer, I.S. & M.R. Becker. *Joy of cooking*. New York: New American Library, 1964.

11.4 Write an algorithm for drawing a picture of a snowman, using a draw tool.

11.5 Translate the following arithmetic expressions into multistep algorithms:

a. The area of a circle = πr^2 .

b. The total cost of ordering through mail order =

$$S(\text{all individual items}) \times (1 + \text{tax rate}) + \text{mailing fees.}$$

A longer example.

Consider the well-known *quadratic formula*:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad (\text{EQ 11.6})$$

used for finding the value of x in an equation of the form:

$$ax^2 + bx + c = 0 \quad (\text{EQ 11.7})$$

Again, any connecting symbol, such as the vinculum or the division bar, acts as parentheses. The single-line “computer-style” representation is:⁴

$$x = \frac{(-b \pm \sqrt{(b^2 - (4 * (a * c))}))}{(2 * a)} \quad (\text{EQ 11.8})$$

The best way to verify that the expressions given in EQ 11.6 and EQ 11.8 are indeed equivalent is to build one from the other. Starting with the original expression, place parentheses wherever the format dictates: around the vinculum, the numerator, and denominator.

$$x = \frac{(-b \pm \sqrt{b^2 - (4ac)}))}{(2a)}$$

Adding all implicit multiplications gives:

$$x = \frac{(-b \pm \sqrt{b^2 - (4 \times a \times c)}))}{(2 \times a)}.$$

and finally replacing the vinculum with `sqrt` and making the division operation explicit gives (EQ 11.8).

Reuse of Parts.

Notice that (EQ 11.8) is really two expressions. The symbol “ \pm ” means “ $+$ or $-$ ”. Therefore, it is really an abbreviation for:

$$x = \frac{(-b + \sqrt{(b^2 - (4 * (a * c))))}}{(2 * a)}$$

and

$$x = \frac{(-b - \sqrt{(b^2 - (4 * (a * c))))}}{(2 * a)}$$

These two expressions share a large common segment. This common segment provides an additional reason for using intermediate values: to represent the common part

3	square root	=SQRT((B1^2)-(4*(A1*C1)))
4	"+" answer	=(-B1+B3)/(2*A1)
5	"-" answer	=(-B1-B3)/(2*A1)

4. The symbol `sqrt`, in this example may be new to you. It is a built-in function, which replaces the square root vinculum. *FORTRAN*, one of the oldest higher-level languages, used `sqrt` for square root and the name has been around ever since. Virtually every computer application uses this notational convention.

where the original values of a , b , and c are placed in A1, B1, and C1 respectively. Obviously even more intermediate values could be used, for example to hold $(b^2 - 4ac)$ or $(2a)$. Intermediate expressions often reduce such redundant computation. The advantage in this reduction is not that it saves computer time, but that it reduces the chance of error and makes the relationship between the two results more explicit.

The two techniques for forcing the order of the steps of an arithmetic algorithm are equivalent. Any arithmetic expression can be converted to an algorithm by either specifying the intermediate steps or by making the order of operations explicit with parentheses. And neither method requires the user to memorize the precedence rules of the system. (In time, every user learns most of the rules of common systems through repeated use, but there is no need to expend energy on this aspect.)

Exercises

- 11.6 Use a spreadsheet to calculate the average of three numbers: 5, 10, and 15. Check your results by hand. (A word to the wise: do this carefully! Simple as it appears, about 50% of all students get this wrong on their first attempt!)
- 11.7 Write an algorithm for computing the average grade in all of your classes this semester. Repeat it for all your grades since you came to college. (Assume all courses receive the same number of credits.)
- 11.8 Open a spreadsheet and perform the calculations from the chapter: meal costs (Table 11.1), both interpretations of outfit costs (Figure 11.1) and the roots of a quadratic equation (EQ 11.6).
- 11.9 Write a single expression (no intermediate steps) for calculating the cost of the meal in Table 11.1.

11.10 You worked 38 hours last week at \$5.57 per hour. The governments withheld 15% for federal taxes and 4% for state tax. You also paid \$3.50 for union dues. Using a spreadsheet, calculate your take-home pay in two ways: once using a single expression and again using an intermediate-value representation.

11.11 Write the implied algorithms for each of the following arithmetic expressions:

$$\frac{3^3}{4 + 5}$$

$$(4 + 7) \times ((8 - 3) \times 6)$$

$$((9 - 7) - 5) - (5 - (7 - 9))$$

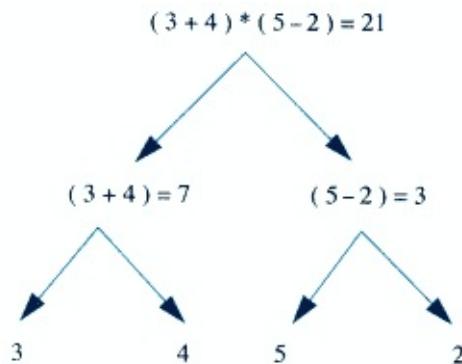
11.2.2

Arithmetic Expressions and Trees

The important concepts of any field seem to reappear across the entire field. Hierarchies (see Chapter 6) are among the most common and therefore important concepts in computer science. They are related to arithmetic expressions in an interesting way: every arithmetic expression has an equivalent hierarchical representation, and that hierarchy specifies an algorithm. For example the expression:

$$(3 + 4) * (5 - 2)$$

is equivalent to the tree:



The position in the tree indicates the complexity of the segment of the expression. Each intermediate node of the tree corresponds to one operation and/or one pair of parentheses. The leaves of the tree are the *atomic* numbers: simple numbers that cannot be broken down further. The root represents the entire expression. The implicit algorithm for calculating the value of each node is:

*If the node represents an atomic value
then that value is also the value of the node
otherwise compute the expression held in the node.*

Calculating the value of the root is equivalent to calculating the value of the expression. Computer applications such as spreadsheets use just such an internal hierarchical representation to

interpret the expressions entered by the user.

11.2.3

Nonarithmetic Expressions

Although arithmetic expressions represent the best known subset of expressions, many others are also important. Complex nonnumeric expressions can also be built up from the basic built-in functions provided by most spreadsheet applications. These functions also are describable by algorithms. For example, two common functions for working with strings of characters are:

concatenation: joins two character strings together to form a single string, and

substring: finds a section within a larger string.

For example, the two operations:

Join the words “bird” and “dog” together into a single word.

and

Find the substring of “bird” which starts at the first character and is two characters long.

might be written as:

```
concat("bird", "dog") = birddog
```

and

```
substring("bird", 1, 2) = bi.
```

Just as numeric operators can be combined in arithmetic expressions, these operations can be combined to form *string-expressions*, each of which has an equivalent algorithm. For example, your initials are derivable from your name. Each initial is a substring of one name and the complete set of initials is composed of the individual initials, so:

```
initials = concat(substring("John", 1, 1),
                  substring ("Q.", 1, 1),
                  substring("Public", 1, 1))
```

evaluates to:

JQP

exactly the same result as the multiline algorithm:

Find the original names.

Find first initial as first letter of first name.

Find second initial as first letter of middle name.

Find third initial as first letter of last name.

Concatenate all three initials together.

Now, consider the problem of writing down the date that your next loan payment is due. Conceptually that is easy something like “exactly one month

5. The exact names of these functions will vary from application to application. For example, concatenation may be called `concat`, `cat`, or `join`; `substring` may be `subst`, or `substr` or `right` and `left`. The exact details will also vary. For example, `substring` is usually a more general operator than `left`. The former may refer to any segment of a long text, but `left` usually refers only to the left-most segment. The manual or online help for your application will provide exact details.

from today.” Specifying the algorithm may be a bit trickier: the algorithm must describe the process for computing a date from any other. That is, if today is

October 15, 1996

the algorithm must be a series of steps that generate

November 15, 1995

Get today’s date (October 15, 1996).

Separate out the parts (e.g., October, 15, and 1996).

Calculate the successor of the current month.

Construct a new date from the three parts: current day, successor of month, and current year.

This is equivalent to the single long spreadsheet expression^{6,7} shown in cell A2 of Figure 11.2.

1	<code>=NOW()</code>
2	<code>=DATE(YEAR(A1),MONTH(A1)+1,DAY(A1))</code>

Figure 11.2
Calculating the Date for a Month from Now

Subalgorithms.

You have seen the operation *successor* before (Section 9.1), applied to letters of the alphabet. Successor can actually be applied to any well-ordered class of item, such as the months of the year, or the days of the week. Figure 11.2 illustrated a common algorithm for dealing with successors that assumes that the items in the set are numbered:

Convert the month-name to a “month-number” (e.g., 10).

Add one to the month-number (e.g., 11).

Convert the new month-number to a new month-name.

This example illustrates two very general properties of algorithms:

An individual step of an algorithm may itself be expressed as an algorithm (here, the *calculate the successor* step can be replaced with a three-step algorithm).

6. Actually, this algorithm will not always work in most spreadsheet applications. Exercise 11.17 addresses the problem, and Chapter 17 provides a solution.

7. The particular syntax will vary from spreadsheet application to application. In this case, the DATE function both converts all of the numeric forms (e.g., 11) to the word forms (e.g., November), and concatenates them together. See your manual or online help for more specific information.

Many algorithms can be generalized (in this case, essentially the same algorithm applies to any ordered series of objects).

Exercises

11.12 Create an algorithm that reformats your name from the form:

John Quincy Public to the form *Public, John Q.* Test your algorithm on a spreadsheet as a multicell algorithm. (Don't forget the period after the initial.)

11.13 Repeat Exercises 11.6 and 11.7 as single-expression algorithms.

11.14 Create the corresponding trees for Exercise 11.11.

11.15 Write an algorithm for specifying tomorrow's date (assuming you know today's). Do the same for tomorrow's day of the week.

11.16 Create expressions to represent your birthday, tomorrow, and a year from today. Test them on your spreadsheet.

11.17 Describe the problem with the spreadsheet representation of the date calculation in Figure 11.2. Under what circumstances will the algorithm fail? Is this a problem with your specific spreadsheet application?

11.18 Write an algorithm for computing the tangent of an angle.

11.19 Pig Latin is defined as:

*For each word
cut off the first character
copy that character to the end of the word
add an 'a' to the very end.*

Thus, Pig Latin for "Pig Latin" is "Igpa Atinla". Convert this algorithm to the language of your spreadsheet.

11.3 Summary

Algorithms are the central subject of computer science. They are essential for instructing machines and also provide the best way for creating clear and unambiguous instruction to humans. The defining characteristics of algorithms include: they are composed of steps, they are unambiguous, finite, and they perform a task. Their use also makes it possible to talk about the process of performing work by giving us a language in which to have the discussion. The concept of algorithm is interrelated with other common computer science concepts ranging from arithmetic expressions to binary trees.

Important Ideas

algorithm step expression

concatenation empirical

12

Generalizing Problem Solutions

Actually, all human problems, excepting morals, come into the gray areas. Things are not all black and white. There have to be compromises. The middle of the road is all of the usable surface. The extremes, right and left, are in the gutters.

DWIGHT D. EISENHOWER

Pointers

*Lab Unit 8: Constructing
Manual: Algorithms Using a
Spreadsheet*

12.0

Overview

Most students entering their first computing class think of problems as if they had the general form:

Here is a specific problem; find the specific answer!

For example, "What is the average of 5, 17, and 92?" It is easy to think that the numeric answer, 38, is the important thing. In computer science, we think of the algorithm as the important part. That is, we engage in problem solving, not question answering. A specific (e.g., numeric) answer is little more than a side effect of proper use of algorithms.

12.1

The Role of Algorithms in Problem Solving

The study of algorithms breaks a single problem (question answering) into two new (but smaller)¹ ones:

1. That same old theme again: “exchange one large problem for two smaller ones.”

Find a solution for the problem in general.

Apply the general solution to the specific problem to find the answer.

For example,

Create an algorithm to find the average of any three numbers.

Now use that algorithm to find the average of 5, 17, and 92.

In fact, it is the general algorithm not the “answer” that is most important in computer science. Why trade in one problem for two new ones? If you only want to find the average of those specific three numbers, why bother creating a general solution? Computer scientists may be tempted to answer with their own version of the famous quote, “Because it’s there!” (George Mallory’s answer to a question about why he climbed mountains): “Because it’s interesting!” Actually, there are at least four very pragmatic reasons for creating general algorithms:

precision

reuse

debugging

sharing the labor.

12.1.1

Precision

Creation of an algorithm including writing it down clarifies the thought process by forcing a problem solver to address each needed step. Putting the intent into words is a powerful tool, both for attacking poorly understood parts of the problem and for seeing flaws in existing solutions. The algorithm provides a formal means for planning ahead and seeing pitfalls.

Most problems are not specified precisely. The question, “What

does this outfit cost?” may mean with tax or without. The process of algorithm creation forces the problem solver to confront that question. Creating an algorithm for the price of the outfit requires asking, “Does that mean ‘including tax?’” Although it may sound like “answering a question with a question,” recognizing such ambiguity in the original request is an essential tool for question answering. Stating solutions in the form of algorithms also forces the problem solver to address the other issues of the solution such as the order of steps and the possibility that it will never be complete.

12.1.2

Reuse

Few problems are as unique as they may initially appear. Many or even most problems will occur again. Suppose you have created a clothing-price spreadsheet. If your roommate wants to purchase a similar outfit, she can reuse the same tool for finding her cost seven if the exact prices are different. After making your initial calculation, you may realize that you cannot afford three full outfits. Possible reactions include, “What about two outfits?” or “What about a

cheaper one?" Or, even if the price was within your means, you may arrive at the store and discover that the prices have gone up to \$4.50 and \$5.50, or that there were several outfits available at differing prices. In each of these cases, the original algorithm can be reused to find the costs. Once you have created the original algorithm, you may simply reuse it as often as you wish. And if that algorithm is represented in a spreadsheet, you need only fill in new values for the independent variables—the spreadsheet does the rest.

12.1.3

Debugging

When things go wrong and they will an algorithm serves as a record of exactly what steps were executed. That record can help determine exactly what is wrong. Conversely, the analysis of the error serves as a tool for creating the new algorithm. For example, suppose you had dinner in a restaurant and left \$10.00 on the table to cover everything—meal, tax, and tip. Suddenly, the server came running after you in a very agitated state. Something is clearly wrong with the amount you put on the table—but what? If you have an algorithm, you can use it to understand your calculations. Did you perform the additions incorrectly or forget the tip? If the algorithm is incorrect, you can fix the error and apply the new algorithm. (Section 12.2 provides additional examples of this technique.)

12.1.4

Sharing the Labor

There are no problems we cannot solve together, and very few that we can solve by ourselves.

LYNDON BAINES JOHNSON

While creating an algorithm does seem to substitute two problems for a single problem, you as the problem solver need not solve both problems yourself. You can ask a helper such as a spreadsheet application to solve one of the problems. You need only specify the

algorithm; a helper or computer can perform the actual calculations and following instructions is what computers do best. Of course the price of sharing the labor is that one individual (the user) must communicate the task (algorithm) to the other (the computer) very carefully. For all practical purposes, the original problem, “finding an answer,” is exchanged for a single new problem, “finding an algorithm.” Usually the solution to the new problem is actually easier than the solution to the original problem.

Specifying Algorithms with a Spreadsheet.

An algorithm can be represented in many forms. Spreadsheet applications provide one of the best methods. The user specifies only the individual steps of the algorithm and the computer performs all of the drudgery. But a spreadsheet is organized as a matrix of cells rather than as a series of steps. *Chapter 13: Programming in Applications* provides a detailed discussion of methods for representing algorithms, but the following very general guidelines serve as a basis for organizing an algorithm on a spreadsheet, while reducing errors and frustration:

Sketch the algorithm first on paper.

Provide a section to enter all initial data (usually at the top).

Specify each step of the algorithm as a function (usually in the middle).

Designate a cell that will hold the final answer (usually near the bottom).

All values should be labeled.

Always test the results.

12.2

Algorithms and Empirical Results

If we begin with certainties, we shall end in doubts; but if we begin with doubts, and are patient in them, we shall end in certainties.

FRANCIS BACON

12.2.1

Experimental Determination of Order of Evaluation

Most of the descriptions of algorithms to this point have been as tools for creating a specific desired result. Science on the other hand is largely concerned with explaining results: why do objects attract, how did life evolve, and so on. The *scientific method* is a controlled methodology of observation for the purposes of understanding the answers to such “how and why” questions.

The computational techniques of *debugging* are essential tools of every computer scientist, and computer users are a subset of the basic empirical techniques of the scientist. Debugging requires testing and explaining the behavior of algorithms, particularly computer programs that do not conform to the expectations of the algorithm builder. Section 11.2.1 suggested that there is little reason to worry about the interpretation of unparenthesized arithmetic expressions such as $3 \times 4 + 5$, because parentheses can always force a desired

interpretation. There are, however, some very real reasons for wanting to know the meaning of $3 \times 4 + 5$. If one person wrote the expression and a second must use it, that second person must interpret the expression in the same way the writer intended. Or a writer who wrote the expression without realizing it was ambiguous now faces the dilemma, “Is the algorithm correct?”

The order of evaluation of arithmetic expressions can be determined by experiment. This experiment is a little more complicated than the “try it and find out” approach of Chapter 1. More complicated problems require more rigorous methods. Very generally, the steps of the scientific method are:

Form a hypothesis.

Devise a test that would determine if your hypothesis is correct or not.

Perform the test.

*If the answer is incorrect,
revise and do it all again.*

For the current example, the original question is:

*Using a specific spreadsheet application, does $3 * 4 + 5$ mean*

- (a) $3 * (4 + 5)$,
- (b) $(3 * 4) + 5$, or
- (c) something else.

Hypothesis.

Suppose the experimenter's initial hypothesis is answer (a). There should be a reasoned justification of the hypothesis. It should be more than a guess. The reason, "It was the first suggested answer," is not a very good reason, but it is a reason. Better reasons include "My algebra instructor used that interpretation," "I used another computer system, in which it worked that way," or "I have been using this system and it seems to behave that way."

Test.

One test for the given question is

Build a spreadsheet.

Perform the calculation.

If the hypothesis is correct, the spreadsheet should answer "27". But if (b) is the correct answer the spreadsheet will produce a different answer (17). Most spreadsheets will evaluate $3 \times 4 + 5$ as 17. In that case, hypothesis (a) would be incorrect and would need to be revised.

Unfortunately, that is the easy part. Hypothesis (c) could also produce the correct answer, "27", (perhaps the spreadsheet itself has a bug and answers 27 for every arithmetic expression; perhaps it is just a coincidence). Additional experiments are still needed to rule out (c) as a possibility.

Revision and Repetition.

It is equally important to conduct the experiment again, with the new hypothesis. Although repeating the experiment with hypothesis

(b) does provide a correct answer, it does not absolutely confirm the hypothesis. Other hypotheses may yield the same answer as (b). Notice one important observation about the above experiments. In each case, we find and rule out incorrect hypotheses. This is a general characteristic of science:

*To prove a hypothesis incorrect is relatively easy;
To prove it 100% correct can be very difficult.*

The general pattern of scientific experiment is

Attempt to show your hypothesis is wrong!

In computer science these observations have been paraphrased as

*There are no correct programs;
only programs with no known bugs.*

Always assume your program is incorrect. Never assume that it is correct. Perform careful experiments to rule out errors. In general, careful experimentation helps (a) avoid errors before coding an algorithm for a specific application by assuring that the designer understands that application, and (b) find and fix errors in existing

algorithms. When correcting an incorrect algorithm, careful experimentation will help the programmer avoid scenarios such as Example 12.1.

Example 12.1 Have a Plan before Fixing. Billie’s spreadsheet produced an incorrect answer. She could tell it was wrong because it said she owed the bank a negative amount of money. Billie randomly selected cells in the spreadsheet, saying “perhaps this formula is wrong.” Each time she guessed, she would change the candidate solution to a new guess.

Needless to say, before long, Billie had not one, but several, incorrect expressions. Fixing the original incorrect expression will no longer generate the correct answer. Billie should follow the golden rule of debugging:

Have a reason and explanation for every correction.

12.2.2

Rules of Thumb and Algorithms

Rules of thumb, as discussed in Chapter 10, are also useful tools for algorithm development:

Use a rule of thumb to predict the approximate expected value.

Use initial test data whose value is easy to verify.

The easier it is to predict the expected value, the more obvious will be any discrepancies and the easier to fix the algorithm. Example 12.2 depicts a “solution” all too common for beginning programmers:

Example 12.2 Evaluating an Answer. Chris creates a spreadsheet algorithm to determine the total cost of a proposed project to put washers and dryers in each unit of a new apartment building. It includes the following (unparenthesized expression):

```
total cost = 387.45 + 513.98 * 14
```

which yields the answer: \$7,583.17. “Good,” says Chris, “no error messages and an answer in the thousands of dollars. It must be right.” Won’t she be surprised when she gets the actual bill for \$12,620.02? Chris’s problem is that her test data wasn’t simple enough. If she had tested the algorithm with the values 300, 100, and 10, the resulting \$1300 total would be clearly incorrect.

Exercises

- 12.1 For each of the following expressions, give two possible values. Predict which your spreadsheet will generate. Compare your prediction with the actual value produced when the expression is entered in your spreadsheet. Note: if you check each expression as soon as you make your prediction, you can use the results to help improve your remaining predictions.

(table continued on next page)

(table continued from previous page)

a. $17 * 12 + 3$

b. $8/2/2$

c. 2^3^2

d. $5 \ 3 \ 1$

e. $5 \ 3 + 1$

12.2 Find other orders of *precedence* for your spreadsheet. Is it “left operation first” or “multiplication first”? Does one rule describe all operations? What about other combinations: addition and subtraction; multiplication and division; multiplication and exponentiation?

12.3

When Algorithms Fail

When a proposed algorithm fails to meet one or more of the requirements of the definition, the failure itself provides information about both the algorithm and the problem. The problem may not be as simple as an incorrectly stated mathematical expression, but it will almost certainly be related to the definition of *algorithm*. Consider two examples:

12.3.1

Infinite Algorithms

Algorithms that do not stop are more common than one might initially guess. Consider the famous mathematics conundrum restated in Example 12.3.

Example 12.3 A computer scientist wants to cross a room, but she must follow the rule: no single step may cover more than one half of the distance remaining. Thus, for a 20-foot-wide room, her steps

could be 10' (half of 20'), leaving 10 feet to go; 5' (half of 10'), leaving 5 feet to go; 2.5' . . . After each step her total distance traveled would be 10', 15', 17.5' . . . The mathematicians watching her from the sidelines claim that she will never get to the far wall.

The ellipsis (. . .) provides a hint of the problem—it is a continuing process with an indeterminate number of steps. Any problem requiring a phrase such as “and so on,” will be difficult to represent as an algorithm. The mathematicians seem to think she will need infinitely many steps. Figure 12.1 on page 208 contains one attempt to solve the problem using a spreadsheet. Each line of the “algorithm” representing one step adds a new positive number, but apparently the total will never reach 20. There always seems to be some distance remaining.

Figure 12.2 on page 208 provides an ingenious but unsuccessful alternative. The designer attempted to avoid the “finite number of lines” restriction by using self-reference: the definition of cell B21 refers to itself. This attempt almost makes sense if the expression in cell B21 is interpreted as

*Add the current distance traveled (B21) to half of the remaining distance
((20 - B21)/2).*

Distance across the room		20 feet	
step #	distance in steps	total distance covered	distance to go
0	10	10	10
1	5	15	5
2	2.5	17.5	2.5
3	1.25	18.75	1.25
4	0.625	19.375	0.625
5	0.3125	19.6875	0.3125
6	0.15625	19.84375	0.15625
7	0.078125	19.921875	0.078125
8	0.0390625	19.9609375	0.0390625
9	0.01953125	19.98046875	0.01953125
10	0.009765625	19.990234375	0.009765625
11	0.0048828125	19.9951171875	0.0048828125
12	0.00244140625	19.99755859375	0.00244140625
13	0.001220703125	19.99877929688	0.001220703125
14	0.0006103515625	19.99938964844	0.0006103515625
15	0.00030517578125	19.99969482422	0.00030517578125
16	0.000152587890625	19.99984741211	0.000152587890625

Figure 12.1
Halfway on Each Step

Unfortunately, this interpretation requires that the value of the cell be known before calculating the value in the cell. That is, the cell must be calculated before it is calculated—a violation of the “ordered” rule. If asked to perform this calculation, most spreadsheet applications will generate an error message stating that there is a *circular reference*. Computer scientists also call this situation an *infinite loop* since the algorithm will run forever.

21	=B21+((20-B21)/2)
----	-------------------

Figure 12.2
A Wayward Calculation

Exercises

- 12.3 Build a spreadsheet similar to the room crossing problem. (You may make the problem easier by using a smaller room, or see Section 15.2.1 for a shortcut.) Extend it further. What happens? Was it what the mathematicians said? Can you think of any possible explanations for the discrepancy?

(table continued on next page)

(table continued from previous page)

12.4 Students of calculus may note that the problem of the computer scientist in Example 12.3 is, of course, the classic calculus concept of *limit*. The limit as the number of steps goes to infinity is actually 20. She could reach the other side if she takes an infinite number of steps but algorithms do not allow an infinite number of steps. Use an algorithm such as the one in the example to see how close your spreadsheet can get to the actual answer. Describe the results. Can you explain them?

12.3.2

Poorly Ordered Algorithms

You have seen that incorrectly ordered algorithms can generate incorrect answers. They can also violate the definition of algorithm by creating a circular reference. Suppose you had a problem involving two values, and you knew that one was twice the value of the other. You might write two related equations:

$$\begin{aligned}\text{first} &= 2 \times \text{second} \\ \text{second} &= \frac{\text{first}}{2}\end{aligned}$$

Any attempt to represent these two equations creates a circular reference. A spreadsheet would generate an error message similar to the one in Figure 12.3. Since each cell is defined in terms of the other; each must be defined before the other can be defined, which is impossible. Therefore it cannot be an algorithm.

Example 12.4 Algebra problems or brainteasers can lead to similar errors:

There are three siblings.

Mary is twice as old as John.

John is five years older than Bill.

Bill is one quarter of Mary's age.

B	C	D	E	F	G
first	=2*C2				
second	=C1/2				

Can't resolve circular references.



OK

Figure 12.3
Circular Reference Error

How old is each sibling?

Suppose you represented this as three equations (and, conveniently, three unknowns):

$$\begin{aligned} Mary &= 2 \times John \\ John &= Bill + 5 \\ Bill &= 0.25 \times Mary \end{aligned}$$

Three equations define three functions. So put it into a spreadsheet:

E	F
Mary	=2*F2
John	=F3+5
Bill	=0.25*F1

Again the spreadsheet will fail because each step requires that one of the other steps be completed first. This is called a *circular reference* because F1 refers to F2, which refers to F3 which refers to F1, which refers to . . . In term of algorithms, an early step must refer to a later step another rule violation.

12.4

Exploring Possible Scenarios

Spreadsheets are excellent tools for exploring alternative scenarios usually called “*what-if problems*”: “What if I made some more money?” “What if the interest

Box 12.4 Algebraists’ Note

You may have noticed that the technique described in Example 12.4 seems to be exactly the most common technique for solving problems of linear algebra: combine the three equations to solve for the three unknowns. That technique seems to work for any resolvable set of equations. Why doesn’t it work here? Actually the technique attempted here is slightly different. The equation in

cell F1 attempts to find a value for Mary all by itself. It is not a combination of the three equations: If you first combine the equations to solve for one value (John = 10) and then base the equations in the other two cells on that value, you can find the solution. Moral: the spreadsheet does the arithmetic not the problem solving.

rate changes?" Given alternative scenarios, the user can use a spreadsheet to predict the outcomes. That information can guide the user's future choices.

Example 12.5 How Many Hours Do I Need to Work? Suppose you need \$1000. And suppose you have two jobs. You like one of the jobs but it only pays \$4.39 per hour. You detest the other, but it pays \$11.31 per hour. Clearly you can get the money faster doing the job that you do not like. But you could also mix the two jobs, taking some of the good along with some of the bad. You might want to ask the questions:

How many hours total must I work doing only the fun job?

How many doing only the not-so-fun job?

*If I want to spend x hours on the fun job,
how many hours must I spend on the not-so-fun one?*

Figure 12.4 shows a corresponding spreadsheet.

1		
2	Amount needed:	\$1000.00
3	Rate of pay for fun job	\$4.39
4	Hours on fun job:	60
5		
6	Earnings from fun job:	\$263.40
7		
8	Amount needed from not-so-fun job:	\$736.60
9	Rate of pay on not-so-fun job:	\$11.31
10		
11	Hours needed on not-so-fun job:	65.13 hours

\$1000.00
\$4.39
60
=B4 * B3
=B2 - B6
\$11.31
=B8 / B9

Figure 12.4
What If You Worked 60 Hours?

“What-if” problems are especially amenable to the use of a spreadsheet and an algorithm. Once the spreadsheet is created, the user need only try various possible combinations of hours:² In addition to problems with a single algorithm and multiple sets of data, you can use “what-if” scenarios to help you understand the impact of new situations as in Example 12.6.

Example 12.6 A Strange Grading Scheme! Instructors sometimes

provide information about their grading criteria. Suppose your instructor's method includes some penalties and some rewards beyond the basic score. Two of the items include:

2. Notice that by placing the rates of pay in cells rather than as part of the functions, the program was made more reusable. The same algorithm can be used for any salary combination.

Late penalty: You will lose 6% per day from whatever score you would otherwise have received (not compound; just multiply the number of late days by 6%).

Extra credit bonus: You can receive an 18% bonus by doing an extra credit problem.

The changes are applied sequentially first the penalty and then the bonus.

Suppose you predict you will earn a basic score of 93 a score you will be happy with. Unfortunately, you also expect to be three days late. That's 18% off. Fortunately, you can make up exactly that amount by doing the extra credit, right? The bonus points would exactly balance the late penalty. Putting the information into a spreadsheet:

12	A strange grading plan		
13	amount of	Score	change
14	penalty or bonus		from original
15	Initial value	93	
16	After penalty	=C15-(B16*C15)	=C16-C15
17	With extra credit	=C16+(B17*C16)	=C17-C15

Cells C16 and C17 correctly capture the needed relationships. The first subtracts a percentage of the score from the original; the second adds the same percentage back in, giving the result:

12	A strange grading plan		
13	amount of	Score	change
14	penalty or bonus		from original
15	Initial value	93	
16	After penalty	76.26	-16.74
17	With extra credit	89.9868	-3.0132

Surprise! Why is that? How could it be? You lost points. In general, always suspect your algorithm and check it. Unfortunately for you, in this case there is nothing wrong with the algorithm or the spreadsheet: the extra credit does not make up for the late score. (Finding the exact cause of the problem is left as an exercise.)

Finally, you can use “what-if” scenarios to examine two situations that differ not just in the initial data. Such situations, like the one described in Example 12.7, require alternate algorithms.

Example 12.7 An Even Stranger Grading Algorithm. The same instructor announced that you have a chance to get lots of extra credit on your next term project.

You get 11% extra for using green ink, and

You get 7% extra for using yellow paper.

The bonuses will be applied sequentially.

A really unusual feature of this scoring algorithm is that each student may choose which bonus is applied first. Making the correct selection may be daunting (the previous example showed that order can make a difference). The best choice of order may not be immediately clear:

Applying 11% first implies that the second bonus will apply to a greater number of points.

If the 7% is applied first, the larger bonus will be applied to a larger amount.

The question, then, is really: “Is it better to take the higher rate first or last?” It is easy to create an algorithm to find the answer:

Double extra credit calculations			
	Green ink first	Yellow paper first	
	bonus	bonus	
8 Initial score	79.00		=C8
9 First extra credit	11%	=C8*(1+B9)	7%
10 Both extra credits	7%	=C9*(1+B10)	11%

Evaluating this algorithm provides the answer:

Double extra credit calculations			
	Green ink first	Yellow paper first	
	bonus	bonus	
8 Initial score	79.00		79.00
9 First extra credit	11%	87.69	7%
10 Both extra credits	7%	93.83	11%

Surprise again! The amounts are exactly the same. In fact, they will always be the samewhich you could verify by trying several different rates and basic scores (Exercise 12.12). (Section 16.4, “Recursion and ‘what-if’ problems” describes an even better way of dealing with this sort of problem.)

Exercises

12.5 Write an English version of the algorithm used in Figure 12.4.

(table continued on next page)

(table continued from previous page)

- 12.6 Rephrase the “what-if” question of Figure 12.4 in terms of the amount of fun-job time as a function of the not-so-fun job time.
- 12.7 Explain the problem in Example 12.6. Hint: Build the spreadsheet. Make sure it produces the same values. Then think very carefully about the calculations. Will other percentages and raw scores yield similar results?
- 12.8 What would have happened in Example 12.6 if your bonus had been added to your score first, followed by the penalty? Can you explain your results? Hint: build the spreadsheet (or better yet, modify the one used for the previous question).
- 12.9 Suppose you use some of your precious money for a major investment (stock, bonds, antiques, chocolate, it doesn’t matter what). On Monday you pick up the newspaper and read that your investment has declined 18% in value since you purchased it. On Friday you pick up the paper again and see that it has gained back 18% since Monday. So you are OK, right? The investment has gained as much as it has lost. Is this correct? Verify your answer.
- 12.10 You have been offered the opportunity to invest in an unusual certificate of deposit at your bank. With this certificate, you deposit your money with the bank for two years. For one of those years, the certificate will pay 13% interest; for the other it will pay only 9%, compounded annually. The really unusual feature is that you get to choose which interest rate is for the first year and which is for the second. Making the correct selection may be daunting: If you select 13% you will have more money at the end of the first year, but earn less thereafter. If you select the lower rate first, you will have less after one year but the larger

percentage will be applied to a larger amount. The question, then, is really: “Is it better to take the higher rate the first year (so that it gets your holdings up sooner) or the last year (so that it will be applied to the compounded sum)?”

12.11 Can you prove that the results of Exercise 12.10 are always true? Hint: The proof requires only facts from high school algebra. Start by attempting to state the relations that exist between the various values and write the arithmetic relation that describes the overall relation.

12.12 Test the last example with three different bonus combinations.

12.13 If the order of adding a percentage bonus does not matter, then perhaps one can save some time by just adding the two bonuses together and then multiplying. (a) Write an algorithm for the calculation based on that assumption. (b) Demonstrate that it works or that it does not work.

12.5 Summary

An algorithm represents a solution to a class of problem. It is often more efficient to solve the algorithm for the entire class than to solve for the single problem.

Creation of an algorithm forces the problem solver to make the reasoning more precise. It also allows for solving a second but similar problem or readdressing the original problem if the data changes.

Important Ideas

scientific	hypothesis	verification
method	circular	infinite
test	reference	loop
“whatif”	algebra	spreadsheet
problem		

13

Programming in Applications

It is easier to perceive error than to find truth, for the former lies on the surface and is easily seen, while the latter lies in the depth, where few are willing to search for it.

JOHANN WOLFGANG VON GOETHE

13.0

Overview

Programming is the process of creating correct algorithms in a form appropriate for a computer application. The term is a much broader term than most people assume. Although *programming* usually refers to the task of creating programs using a classical computer language such as *Pascal*, *Modula*, or *C*, the process of creating a program or algorithm is essentially the same for any application and the term *programming* is equally applicable to the task of creating a good spreadsheet or database. *Programming* is really a more formal word for the task of the designer. Even effective use of representational tools, such as word processors or graphics programs, requires aspects of programming. Programming also refers to a much broader task than most new programmers believe. Although algorithm development is the primary or most important step in programming, it is nonetheless only one step in that much broader task. The “bigger picture” includes:

problem definition

algorithm development

coding

documentation

debugging, testing, and verification.

Each of these is an essential and important tool for creating useful computer tools.

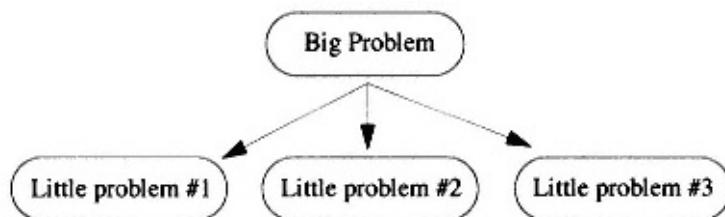
13.1

The Programming Process

The central and most important step in creating a program is *algorithm development*, the logical translation of a problem into an algorithm to solve that problem. Although Chapter 11 claims that any well-defined problem has at least one

corresponding algorithm, some people may find it difficult to find or create that algorithm. It is clearly possible to calculate future positions of astronomical bodies (e.g., many calendars indicate the dates of full moons). Yet, to most people, it is not even clear how to begin such a calculation. Even the details of a generally familiar process often are not clear. Everyone must pay taxes to the IRS, and those taxes are based on income: the higher the income, the higher the tax. But few people understand each step needed to figure out their exact obligation. The major task of programming is developing the algorithm.

Chapter 6: Hierarchical Organization of Information describes hierarchical data structures, such as trees and outlines. Creating an algorithm similarly reduces a complex problem to several smaller, subordinate problems, each reflecting an individual step. Both algorithms and algorithm design are thus *hierarchical* or *top-down*.



The algorithm is hierarchical because each step has substeps. The process is *top-down* because it starts at the top (root) goal and works downward toward individual subgoals. Although a problem or function may seem very complex on first inspection, the lowest-level individual steps are always very understandable. The general guideline

Two small problems are almost always easier to solve than one large problem

provides the basis of the technique, *divide and conquer*. Breaking a problem into two separate problems is a fundamental problem-solving tool. If the problem is not easily understood, divide it into steps or subproblems and repeat the entire development process on

each step. Therefore, algorithm development is also an *iterative*,¹ or repetitive, process, consisting of a series of steps repeated (iterated) several times.

For each subproblem
if the subproblem is simple
then find an algorithm for that subproblem;
otherwise divide the problem into subproblems
repeat these steps on the subproblems.

1. *Iteration* is a major effort-saving concept and is one of the most fundamental techniques of computer science. In fact, Chapter 15 is devoted entirely to that single concept.

Several common techniques can be applied at each level of this iteration, including:

Understand the problem.

State the major goals.

For each goal:

describe the known information,

describe the needed information,

describe the difference,

attempt to create an algorithm representing that difference.

Example 13.1 illustrates both the top-down and iterative aspects of the program development process from the problem definition through to verification.

Example 13.1 Household Expenses. Chris, Pat, and Joe are roommates who share household expenses equally. Whenever anyone makes a purchase, the receipt goes into a basket. At the end of the month, they dump the receipts on the table and figure out who owes whom how much money. Find an algorithm they can use each month to determine how much each person owes (or how much that person is owed).

Level 0.2 At the start, the description is a problem, not an algorithm. Start by understanding the problem and stating the goals:

State what is known: the individual purchases.

State the major goal: find out how much each person owes.

Understand why the goal is needed.

Level 1. At this point the description is still only a goal and some initial data. Ask “What is the difference between what is known and what is needed?” Use the answers to generate substeps and definitions:

Each person who has paid less than the others owes money.

Each person who has paid more than the others is owed money.

Level 2. Divide the problem into subgoals. Attempt to understand each subgoal. The terms at Level 1 are not very specific: How much exactly does each person actually owe? What does “more than the others” mean exactly? How do you find that number? What is the status of a person who paid more than one person, but less than another? State the definitions more precisely:

2. Recall that computer scientists tend to describe the initial conditions as step 0. Nothing has been calculated yet. Zero is a good designator, since the first calculation will be called step 1.

*Who paid more than the average of the three?
 Each person owes the difference between
 the average and the amount actually paid.*

This implies that you must calculate the average before finding the amount owed. In addition, the difference must be found for each of the three individuals:

*Record individual purchases by each person.
 Find the average paid.
 For each person:
 find the difference between what that person paid and the average.*

Level 3. Average is usually calculated using the function:3

$$\text{average} = \frac{(\sum \text{individual items})}{\text{number of individuals}}$$

In this case, the total purchased is the sum of all the purchases, which, in turn, must be the sum of the individual total purchases. Therefore, before calculating the average:

*For each person:
 add up all of that person's purchases.*

Then,

$$\text{average} = \frac{(\text{Chris's purchases}) + (\text{Pat's purchases}) + (\text{Joe's purchases})}{3} \quad (\text{EQ 13.1})$$

The algorithm now becomes:

*Record individual purchases.
 For each person:
 find the total paid by that person.
 Find the average paid, using (EQ 13.1)
 For each person:
 find the difference between the average and what that*

*individual
paid.*

3. Computer scientists and mathematicians use the Greek letter sigma, S, to mean “the sum of all the following.”

Level 4. These steps expand trivially. Difference is found by subtracting the average from that person's total:

$$\text{amount owed} = \text{average} - \text{person's total}$$

The algorithm also serves as a tool for interpreting negative results. The problem statement indicates that some people would be owed money, but the algorithm calculates how much a person owes.

*If a person paid more than the average
then that person must be owed money, and
the difference, average - person's total, will be
negative.*

That is, a negative difference indicates that the person is owed rather than owes. Use this observation to expand the algorithm:

*Record individual purchases.
For each person:
 find the total paid by adding up that person's purchases.
Find the average paid, using (EQ 13.1).
For each person:
 find the difference between
 the average and what that individual paid,
 (record both positive and negative results).*

Level 5. Finally note that “for each person” implies three separate calculations, one for each housemate:

*Record individual purchases.
Find the total paid by Chris, by adding up Chris's purchases.
Find the total paid by Pat, by adding up Pat's purchases.
Find the total paid by Joe, by adding up Joe's purchases.
Find the average paid, using (EQ 13.1).*

Find the difference between the average and what Chris paid.

Find the difference between the average and what Pat paid.

Find the difference between the average and what Joe paid.

This final version of the algorithm, represented as a spreadsheet, appears in Figure 13.1(a) on page 222, producing numeric results as in (b). Each cell contains a function corresponding to one step in the algorithm. In fact, the functions are simply the arithmetic restatement of the English description. Since Pat owes a negative amount, Pat must be owed money. The other two owe positive amounts and therefore they each actually owe money apparently to Pat. In addition to the development of the program itself, the programming process includes both documentation and testing, as described in later sections.

1 Expenses for the month of October					
2 who	purchases			Totals	
3 Chris	\$5.00	\$20.00	\$13.00	\$92.00	=B3+C3+D3+E3+F3
4 Pat	\$437.00				=B4+C4+D4+E4+F4
5 Joe	\$37.00	\$37.00	\$37.00	\$37.00	=B5+C5+D5+E5+F5
6					
7 average					=(G3+G4+G5)/3
8					
9 differences	Chris	=G7-G3			
10	Pat	=G7-G4			
11	Joe	=G7-G5			

(a)

1 Expenses for the month of October					
2 who	purchases			Totals	
3 Chris	\$5.00	\$20.00	\$13.00	\$92.00	\$130.00
4 Pat	\$437.00				\$437.00
5 Joe	\$37.00	\$37.00	\$37.00	\$37.00	\$185.00
6					
7 average					\$250.67
8					
9 differences	Chris	\$120.67			
10	Pat	-\$186.33			
11	Joe	\$65.67			

(b)

Figure 13.1
Algorithm for Calculation of Household Expenses in a Spreadsheet

13.2 Problem Definition

If I hired you to dig a hole for a septic tank in my back yard, but you dug a hole in my rose garden, should I pay you anyway?

RAY SCOTT

Problem definition comes first. Yet, it is the most overlooked step in the program development process. In fact, misunderstood problems are one of the most fundamental sources of error. Problem definition is always the first step in the programming process. Make sure you fully understand the problem. Developing a "correct" solution to the wrong problem can require as much time and effort as does any solution to the right problem. And that represents unrewarded time and effort; solutions to incorrect problems seldom receive much credit. The hole digger above is not likely to get paid for the hard work performed. A motorist who diligently drives under 55 in a 35-mile-an-hour zone will receive little sympathy

from an arresting officer, when she explains that she thought the speed limit was 55. Similarly a computer program that solves the wrong problemno matter how elegantlyis useless.

13.2.1 *Ask Questions*

*I keep six honest serving-men
[They taught me all I knew];
Their names are What and Why and When
And How and Where and Who.*

RUDYARD KIPLING

In colloquial terms, problem definition means: “Ask a lot of questions!” Generally the creator of a problem and the programmer who creates a program to solve that problem are not the same person. It is the programmer’s job to ask questions of the problem poser. In the classroom situation, it is the student’s job to ask questions of the instructor. Even problems that you pose for yourself (e.g., “I’d like to know how much interest I will pay on my mortgage.”) have criteria defined by others, such as “how is monthly mortgage interest calculated?”

Ask questions of any useful information source. Useful information sources are everywhere. In particular ask:

the problem poser

people familiar with the problem or the situation

your friends

even yourselfas an impartial observer.

Surprisingly, even asking someone who probably does not know the answer may help. The act of putting the question into English is a useful problem-solving technique because it forces you to clarify vague points. It forces you to think about the definitions as you select the words to describe the problem. You will find that by the

time you finish asking a question you have defined or identified several additional aspects of the problem.

13.2.2

Input-Output as Defining Factors for Problem Definition

Describe the problem in terms of the input and output requirements.

What information is needed (output)?

What is available (input)?

Is the input information realistic? Will it be available when needed? (A tax computation program should not ask the user: “How much do you owe in taxes?”) Does the output satisfy the needs of the user? Ask: “What are the logical relationships between the input and output?” Extra care paid to the details of input and output will more than pay for the effort. Careful definition of input and output requirements helps form a new question:

How can I generate the output using only information held in the input?

This question actually starts the algorithm development process by creating three subproblems:

Get the input information.

Calculate the results.

Display the results.

each of which can be attacked and conquered separately. The divide-and-conquer approach begins immediately and with little effort.

13.2.3

Other Definition Tools

Much information already exists in printed form. Read published guidelines for the task (e.g., the tax return guidelines); dictionaries (look up the definitions of any words in the problem that you are not absolutely sure of; e.g., some people pay too much tax because they do not know the difference between “income” and “taxable income”); and the manual for the application.

Problem Definition’s Bottom Line.

Problem definition must come before all else. Do not start developing an algorithm until you know what the algorithm must accomplish.

13.3

Algorithm Development in Context

Algorithm development, as described in Section 13.1, is really the task of creating an algorithm from a well-understood problem before translating that problem into the code needed by a computer application yet another example of the divide-and-conquer approach. It is really the task of translating the problem into a form amenable to programming. It holds the central role in the programming process, both logically and chronologically. Obviously it is more efficient to develop an algorithm for a well-

understood problem. It is also easier to think about the algorithm without distractions created by the code.

13.3.1

Use Top-Down Design

Use the top-down design process described in Section 13.1. You may even find that it helps to design without the aid of a computer. That will remove the temptation to begin coding before you are ready.

Start with the Input-Output Definitions.

Carefully describing the available input and the desired output, as described in Section 13.2.2 can almost always help start the design process. It divides the algorithm into at least three parts as well as restating the goal into one of transformation.

Level of Detail.

The top-down process successively refines the algorithm, creating steps with more and more detail. When can you stop? You can stop the

process when you can see how to write the steps unambiguously in your application. In a spreadsheet, each step should be small enough so that you can code it in a single spreadsheet cell. And remember: there is no reward for packing large amounts of code into one cell. In fact there is a penalty: it will be harder to debug.

13.3.2

Use Pseudocode

Write your algorithm in *pseudocode*—structured English statements before attempting to code the algorithm for your computer application. The term *pseudocode* implies that it is not real computer code, but “just” English. For example, to find the cost of a mail-order purchase, you might write:

(Find the individual costs.)

Find the total of all items.

Multiply that total by the applicable tax rate.

Add the result to the previous sum.

Add the shipping charge.

Display the result.

To many, this seems like a lot of extra work, but the effort more than pays for itself as problems become more complex. Writing pseudocode separates the task of developing the algorithm from the task of coding for the application. Keep the two questions separate:

How do I solve this problem (step)?

How do I represent that action in the code of my application?

Many programmers are overwhelmed because they attempt to code an ambiguous or poorly thought through step into a rigorous and exacting application language. Pseudocode algorithms can also serve as the basis of much of your documentation (see Section 13.5).

What Makes Good Pseudocode?

There is no single definition for pseudocode, no concept of correct or incorrect pseudocode. So how do you start creating it? The following guidelines are by no means absolute, but they should help.

1. Pseudocode should be English-like. It should be easy for humans to read.
2. It should be algorithmic: it should show ordered steps.
3. It should remove ambiguity.
4. The target level of detail should be one pseudocode statement per computer instruction (cell definition in the case of a spreadsheet).
5. Each word should be clear or well defined. In fact this is the essence of pseudocode: it helps the programmer see the incomplete or ill-formed details.
6. Beware of “fuzzy” words such as: “some,” “a,” “an,” or “one.”

Box 13.1 When Natural Is Pseudo

It may seem like an odd commentary on computer science that its practitioners at one point focused so completely on the final code as represented by a programming language, that the original natural language is called “pseudo.” The importance of fully specified English language algorithms was not really understood for many years. Unfortunately, new programmers often mimic the evolution found in the field as a whole: they do not discover the importance of separating the algorithm from its coded representation except through the “school of hard knocks”: they only learn through their mistakes.

In reality, the term implies that a structured representation can be precise without bogging the writer down in syntactic details. It allows the programmer to focus on the important aspects of the problem. Pseudocode can be as rigorous as a “real” computer language.

7. If you use words such as “the” or “it,” you should be able to find the antecedent.
8. Mathematical notation is almost always acceptable.
9. Algorithms in this text are all written in pseudocode. You may want to use them as examples.

13.4 Coding

Many new programmers confuse the coding step with the entire programming⁴ process. *Coding* is the task of translating the written algorithm into the language of a specific application. The ultimate language in which the program will be coded has only a very minor influence on the definition of a problem or the algorithm development. The application does have a major influence on the

coded product. But it is the mundane detailsthe final representation, the available documentation tools, debugging aids, and so onthat differ between the products of competing vendors. Coding does appear to be the focal point of the programming

4. This misconception is very widespread. Even many textbooks seem to get it wrong: some often have titles and chapter organizations that seem to place the entire emphasis on the details of an applications language. But, under closer scrutiny, even these books devote considerable discussion to the other aspects of programming.

task, and it is certainly the most readily identifiable step. Yet, it is actually one of the most mechanical and least time-consuming steps. The result of the coding process will, of course, look very different for different applications, but the process is essentially identical:

*For each step of a refined algorithm
translate it into a single step in the appropriate language
representation.*

It is at the coding step that the divide-and-conquer approach and the extra effort of writing pseudocode starts to pay off. Well-written pseudocode can be translated almost directly into the language of any appropriate application. If the translation for one step is not clear, it must be for one of two reasons:

the algorithm has not been specified in enough detail
the programmer does not know a syntactic detail of the application.

In the first case, iterate the algorithm development process one more step. In the other case, look the command up in the user's manual.

Code Bottom-up.

Although algorithm design is a top-down process, the actual coding will usually be a bottom-up process. As you code each line of your algorithm, check that line to make sure that it is correct. In the case of a spreadsheet that simply means

double check what you actually wrote
check the values that it produces.

The best time to make these checks is as you code the step when the intended meaning is still fresh in your mind. To check it later requires that you "reunderstand" what you were doing. Remember:

One of the major causes of error is correct algorithms incorrectly coded.

The more detailed the English language pseudocode, the easier the coding task, and the less likely it is that there will be errors. The more careful the checking, the fewer errors will survive.

Procedural Abstraction.

The proverb

*If you want it done right,
do it yourself!*

is apt in life, but not necessarily in programming. Because of the common need for many functions, the creators of most applications build those functions into the applications. Use these built-in functions wherever possible and appropriate. Not only will that use save you the time of figuring out how to construct the function, but if a function has been used by many others, it is likely that any (most) errors have already been discovered. Once again the concept of reuse aids problem

solving. (*Chapter 19: Abstraction, Abbreviation, and Macros* elaborates on this concept.)

Coding in a Spreadsheet.

For spreadsheet users coding becomes the translation of pseudocode into cell definitions. It is the selection of the appropriate representations for the individual actions. If the pseudocode is well written, coding is a very straightforward process. Coding also involves issues of documentation. For example, filling in a cell requires that the cell have a location. But that requires formatting the page, which is really part of the documentation process.

13.5

Documentation

Documentation provides information for humans—the designer and the user; it is not for the computer. To many this seems paradoxical. Documentation helps the humans understand what the computer is doing, or more accurately, what the programmer intended the computer should do. It also helps the user understand what information the program needs. As you will see, getting the program right is the major task of the programmer. Documentation helps make sure it is right.

The basics of documentation were laid out in Section 10.4. As you develop more complicated algorithms, you will need more documentation.

13.5.1

Document as you Code

Do not wait until the program is complete to begin documentation. Concurrent documentation provides an essential tool for the other aspects of the programming process. The documentation should describe what you intend for the code to accomplish. The difference

between your intent and what the code actually does is the primary guide for debugging.

13.5.2

Document for the Concerned Parties

Recall that the person who defines a problem and the person who writes the code are normally two separate people. Documentation serves two separate parallel roles:

clarification of the input and output values (for the user).

clarification of what the program is doing (for the designer).

For the User.

If the user cannot correctly interpret the input requirements, there is only one result: errors. For example: almost all ATMs allow the user to withdraw cash; all of those insist that the amount withdrawn be an integral number of dollars (in fact, they usually demand multiples of five dollars). Some but not all ATMs ask the user to enter the cents as well as the dollar amount. Since most machines automatically add the decimal point and the dollar-sign, the user who wishes to withdraw \$100.00 should either enter:

100 if cents are not required

10000 if cents are required

In either case, an inattentive user at an unfamiliar ATM can receive a major surprise after typing the wrong representation for \$100. In a spreadsheet, instructions should clearly indicate to the user:

where input is needed

how it should be represented

restrictions on the input such as the legal value range.

Figure 13.2 requests the ATM information. The input location is clearly indicated by the box; the instructions specify that input is needed and how it should be represented.

Unclear output can be just as confusing. Suppose a program provides information about a loan that the user wishes to make. If the program represents the interest rate as “.08”, does that actually mean 0.08%, or does it mean 8%? (Probably the latter, because the former would be a record low.) It could even mean 0.08% per month rather than per year. No matter how accurate the result, it is useless if the user misunderstands it.

For the Programmer.

When a program produces an incorrect result, there is generally a discrepancy between what the programmer intended and what the programmer actually said. The documentation should explain what was intended and why the steps are there. Such documentation helps the programmer pinpoint discrepancies between intent and result.

One of the most common errors that programmers make is adhering to the erroneous belief that they can remember exactly what every

step of the code does. Human memory is just not that good (e.g., Miller's short-term memory observations mentioned in Section 3.2). Recall that one of the reasons for creating an algorithm (rather than just solving a specific narrow problem) was reuse. If you want to use the spreadsheet next year, you will not remember the details. If a friend borrows it, she will not know the details. Documentation helps spot errors long after the original thinking is forgotten.

B	C	D	E	F
---	---	---	---	---

How much money do you wish to withdraw?
Include the decimal point, and cents.
(Total must be a multiple of 5 dollars.)

Figure 13.2
User Documentation

Compared to traditional programming languages, spreadsheets are both easier and harder to document. On the one hand, a spreadsheet is not a blackbox. All formulae and all intermediate data are always available for inspection. The programmer or user can always check how an individual value was calculated. On the other hand, the application provides no specific documentation format; it is up to the programmer to develop a clear method.

13.5.3

Some Documentation Guidelines

The general rules for documenting a spreadsheet are essentially the same as those for representing relations and functions:

Distinguish input from derived data (see Section 13.2.2).

Format for clarity. Make it as easy to read as possible. For example, line up numeric cells vertically.

Label all information.

Place important items in positions of prominence and/or highlight them.

Format the individual cells to reflect their use. For example, if the cell represents a monetary sum, then format it with the appropriate symbols (e.g., "\$").

Generally place the input at the top (or left), intermediate calculations in the middle in order of calculation, and results at the bottom (or right). That is, do not calculate line 3 from lines 2 and 17.

Include intermediate values where possible. They should be clearly differentiated and not clutter or obscure the answer.

Describe why a step is there, rather than what it does.

It is hard to overdocument a program or other living document.

When in doubt, document! Documentation just may be the programmer's single best defense against errors.

13.6

Debugging, Testing, and Verification

It may seem that an inordinate amount of the effort in creating a program has really gone toward errors: preventing errors, detecting errors, fixing errors, and so on. But the attention is deserved.

Programmers may be the only people who spend most of their time fixing their own errors and still keep their jobs. In fact the last three aspects of programming are so important that they have been segregated in a separate chapter, dealing exclusively with errors.

Exercises

13.1 Examine the algorithms you wrote for Exercises 11.1, 11.2, 11.4, and 11.7. Compare them to the rules for pseudocode given in this chapter.

13.2 Examine the algorithms you wrote for Exercises 11.11, 11.12, 11.15, and 11.18. Compare them to the rules for pseudocode given in this chapter.

(table continued on next page)

(table continued from previous page)

- 13.3 If S or summation is so important in math and computer science, you might expect spreadsheets to provide a tool to calculate sums. Find and test that tool in your spreadsheet.
- 13.4 Take several spreadsheets you have worked on and reformat them, providing all of the needed documentation.
- 13.5 Write the programs described in Laboratory Unit 8 as spreadsheets. Document them appropriately.

13.7

Physical Appearance

The physical appearance of a document has no direct impact on its correctness. But it does have a significant impact on both the programmer's ability to implement a correct algorithm and the confidence the user places in the result.⁵ Attention to appearance and detail actually reduces the number of errors. Develop habits that lead to programs that are both easy to read and easy to debug. Following the guidelines below will help you develop easy-to-read-and-debug spreadsheet documents. The guidelines are just that, guidelines; they are not absolute rules. There will be situations when they are not the most appropriate, but start out by following the guidelines. Although many of the rules provided here may appear to be dogmatic and to require extra time, it is time well spent. The total time spent creating a program that runs correctly will actually be shorter if you follow these rules diligently.

The time to follow these guidelines is while you are writing the program: Never write a sloppy program, intending to "clean it up later." The intent of demanding programs that look good is that paying attention to detail forces you to get both the general algorithm and the "little" points correct.

13.7.1

Whitespace

In many of the pseudocode algorithms in this text, blank lines separate logical sections of the document. Just as blank lines can separate or group the sections of pseudocode, such spacing helps emphasize the structure of the program by indicating that some blocks of statements are more closely associated than are others. Similarly, indentation of a group of statements shows that they form a logical block that is subordinate to the preceding or following segments.

Whitespace is just as important in a spreadsheet. It is essential for clear and uncluttered spreadsheets, that are easy to read and understand. Do not be afraid of using too many cells. Use blank lines to break a program into logical segments (much like paragraphs). In general, include whitespace to separate

5. And, for better or worse, the opinion of a grader on an assignment.

logically separate portions

user input from intermediate values from final output

blocks of comments or instructions from blocks of code.

An Exception.

A document that almost, but not quite, fits on a screen may be easier to read if it is shortened slightly. Balance the need for distinguishability with the need for a comprehensive view.

Borders.

Cells can also be set off by careful use of border formats. Use them to distinguish selected cells (e.g., input values) and make them easy to find.

13.7.2

Comments

Comments are essential all programs should be well-documented.

Distinguish Comment from Data.

Make the difference between comment and data or function clear. Although this should be easy for the careful reader, the difference should be apparent without forcing the reader to be exceedingly careful. Do not put all of the responsibility on the end-user.

Comments can be distinguished from data by font, style, position, or because they look like prose. Comments should include:

1. Heading. Begin all programs with a brief heading, which includes the author's name and a brief description of the problem and how the program works.
2. Input/output. Describe the input required by the program and the output that it produces.
3. Why. Place an emphasis on why a statement or block of statements is there rather than precisely what it does. What a

statement does can be seen by inspection; why the programmer wanted it to cannot! Compare the following two segments of documented code:

=D2+D3	add D2 and D3 together
=D2+D3	find total income

The former adds nothing that any knowledgeable programmer cannot see directly from the cell definition. The latter both indicates why the statement is included and reminds the reader of the use of D2 and D3.

Organization.

Arrange comments and data neatly. When comments occupy several lines, each line should start in the same column.

13.7.3

Organization and Readability

Intermediate Values.

Display intermediate checkpoints if there is any doubt as to the accuracy of the answers.

Output.

Make sure that all results are clear, labeled, well-organized, and readable. Tables and lists of values should be well structured (e.g., in neat columns).

User Instructions.

Provide the user with complete information about what the program does and how to use it. Useful messages should be printed to inform the user what values are needed and in what form.

Remember, the user may not want to look at the actual code!

Indentation.

Indent logically subordinate sections of your spreadsheet. Take control of the appearance of structure: select a format that reflects your intent.

Example 13.2 The restaurant bill illustrates several style techniques. Two columns are used for totals so that the reader can tell what items have been added together. Borders separate groups of items. Underscores indicate summations. Grouped items are indented.

Bill for Joe's Bar and Grill

Restaurant bill

steak dinner	\$12.95
lobster special	\$14.50
onion soup	\$2.95
chocolate suicide cake	\$2.50
Restaurant total	\$32.90
Tax on restaurant bill	\$2.47

Bar bill

2 aluminum city beers	\$5.00
wine cooler	\$2.95
after dinner sherry	\$5.50
Bar Total	\$13.45
Total for restaurant and bar	\$48.82
Gratuity (15% of all items except tax)	\$6.95
Grand total	\$55.77

Violations.

Violation of these style rules should reflect a deliberate and conscious action. There are valid reasons for violating the rules, but you should be very aware that your action may cause confusion. Weigh the pros and cons before

violating these rules. That's what guidelines are for, after all: to provide a generally safe path.

13.8

Summary

Programming refers to much more than just the task of coding an algorithm. It also includes the essential and important steps of algorithm development, problem definition, documentation, testing, debugging, and verification. These are skills that really come through practice.

Important Ideas

program code definition
documentation format whitespace
comment debugging testing
verification

14

Dealing with Errors

As soon as we started to program, we found to our surprise that it wasn't as easy to get a program right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life was going to be spent in finding mistakes in my programs.

MAURICE WILKES¹

14.0

Overview

Errors can be a plague to programmers and designers. Fortunately they need not drive you to despair. Careful planning before and thorough analysis after can greatly reduce the impact of errors. If you understand what causes errors and what the symptoms are, you have a head start in the process of preventing, finding, identifying, and removing the most common and troublesome problems.

14.1

Sources of Error

Although neither writing nor coding an algorithm is difficult, errors almost always seem to creep in. Finding and fixing these errors a process called *debugging* is actually the most time-consuming task of *programming*. Programmers especially new programmers often find, as Wilkes did, that debugging takes more time than all the other steps put together.

To err is Human

they say. Humans are indeed fallible. That is one reason why we use computers to eliminate computational errors. But computer results do contain errors. Unfortunately, the human writes the algorithm. Therefore, the program is subject to error. Human errors and therefore incorrect results come from any of several

sources:

incorrect algorithms,

incorrect representation of an otherwise correct algorithm (e.g., a “typo”),

1. Maurice Wilkes (b. 1913-), a British computer scientist, designed and built *EDSAC* (Electronic Delay Storage Automatic Calculator).

Operational in 1949, EDSAC was the world’s first full-scale electronic computer that actually worked.

incorrect interpretation of the results, and
incorrect values for independent data.

The first two of these occur more in the design phase and the last two are really problems for the end user. Debugging to fix those errors is an integral part of the programming process.

Incorrect Algorithms.

Getting the correct algorithm is the central role of programming (as seen in the previous three chapters). Errors such as those in temperature conversion (Example 10.2) or for the cost of clothing (Section 11.2.1) are examples of incorrect algorithms. The major sources of error in the algorithms come from errors in the design process:

incorrectly understood problem (e.g., misunderstood initial conditions or desired result) or

incorrect order of operations (e.g., performing arithmetic steps incorrectly) often caused by incorrect assumptions.

Incorrect Representation of an Otherwise Correct Algorithm.

This class of error includes typos (programmer thinks one thing but writes something else):

”3 + 41” instead of “3 + 14”;

incorrect representation of expressions (e.g., programmer thinks “3 + 4 * 5” will be interpreted as “(3 + 4) * 5”.

The class can be difficult to see since humans tend to see what they intended rather than what they actually wrote. Remember to read what you wrote carefully. Never allow yourself to assume that you actually typed correctly.

Incorrect Interpretation of the Results.

Remember that the output of a program is just that: the results of

the algorithm you wrote. You need to interpret the results carefully. For example, suppose you wrote a program that tallied the results of an election. One common result expected of such programs is a calculation of the percent of the votes received by a candidate. If the output in the appropriate location is “.15,” does that mean “0.15%” or does it mean “15.0%”? Incorrectly interpreting the results of the algorithm can produce answers as invalid as those produced by an incorrect algorithm.

Many interpretation errors involve a more subtle error in the conclusions that may be drawn from a calculation. For example, the error in Example 14.1 actually appeared in an undergraduate thesis.

Example 14.1 Paul Programmer, a student at The Federal Institute of Technology was interested in the use of spreadsheets by students for completing their homework. He surveyed 100 randomly selected students and found that 80 male students used spreadsheets regularly, while only 20 female students did so. Paul concluded that males were four times more likely to use spreadsheets than women. Unfortunately Paul did not include in his interpretation the small

detail that the FIT student body has 1600 males and only 400 females. That is, there are four times as many male students as female: exactly the same ratio as he found for spreadsheet users. Paul had correct data but did not make the correct conclusion from that data.

Incorrect Values for Independent Data.

This is the famous *garbage in; garbage out!* (or *GiGo*) cliché. No matter how well tested the algorithm is, the results are wasted if the data is incorrect. Bad data can result from many sources, including:

bad assumption (user assumes that 15 means “15%”, not “15 persons”).

bad data in the sources (e.g., student enters data from the professor, but the professor wrote the incorrect values in the handout).

typos (e.g., student types the wrong value).

14.2

Removing Error

14.2.1

The Debugging Process

Like the development of a complete algorithm, making sure it is correct is also an iterative process:

Until you are sure you have the right answer:

Specify the algorithm using the top-down design process.

Test the algorithm.

Find any errors.

If there are errors,

figure out what you think caused them.

fix the errors.

repeat the entire process.

One could call this an “algorithm for debugging algorithms.”

The last three aspects of the programming process: debugging, testing and verification, are inextricably tied together. Generally *debugging* refers to getting the bugs out, while *testing* refers to looking for bugs. *Verification* refers to systematic approaches to convincing yourself that the algorithm is correct. Clearly when you find a bug, you need to remove it. The big question is: “How do you fix a bug once you find it?” Even this problem has two subproblems:

What is the problem?

What will correct it?

14.2.2

Debugging and the Scientific Method

Every time a programmer debugs a program, she practices the scientific method (see Section 12.2). Restating the steps of that method in the debugging context provides important guidance for what to do when you find an error:

Form a hypothesis about what could cause the error.

*Determine a test that would determine if your hypothesis is correct
or not.*

Perform the test.

*If you are right about the source of the error in the algorithm
then fix the algorithm and test it again;
otherwise revise your hypothesis about the error and try that
again.*

Notice that the method requires that you think about the error before fixing it. Changing the code without a thought-out justification will likely result in code with not one but several errors (as poor Billie found out in Example 12.1). Similarly, looking for just any mistake in the codewithout reason to believe it could cause the observed problemis not nearly as productive as looking for a mistake of a specific form.

14.3

Anticipating Errors or Anti-Bugging

The best way to fix errors is to avoid them altogether. If that is not possible (and it seldom is), you should write code that anticipates possible sources of error. When you separated the algorithm development from the coding step, you were actually anti-bugging: the smaller problems are not as likely to create errors. Errors are more obvious when the algorithm is easy to understand. Because errors are so common, make a plan for their detection and removal.

Intermediate Values Revisited.

Section 11.2.1 suggested using intermediate values for clarity. They provide an additional service: checkpoints for recognizing and finding incorrect values. Any nontrivial algorithm will have intermediate values. In the roommate example (Figure 13.1), the individual totals and the average are intermediate values. Although

it is often possible to reduce a series of arithmetic expressions to a single expression, it is not necessarily a good idea. Multiple simple expressions are clearer and easier to evaluate than more complicated ones. In the roommate problem, the amount Chris owed could have been calculated using the single relation:

$$C9 = \frac{(B3 + C3 + D3 + E3 + F3) + B4 + (B5 + C5 + D5 + E5 + F5)}{3} - (B3 + C3 + D3 + E3 + F3)$$

While correct, this equation is not clear. It does not reflect the steps of the algorithm as derived in the top-down process. Most importantly, it is very difficult for the user to be sure it is exactly right. It does require fewer cells, but “cells are cheap” (see Section 14.6). Chapter 9 emphasized the importance of checking answers for feasibility to remove the obvious errors. More complicated problems

require more diligent checking. But algorithms with checkpoints help the programmer find the error. For example:

1. Two errors may offset each other, providing a reasonable looking answer. Suppose both the totals and the averages were too low. In fact, if all totals were too low, the average would certainly be too low also. This situation could easily result if not all purchases were added up. The difference between them could appear reasonable. Errors in the intermediate totals are often easier to see. For example if one person purchased two items, but only one was included in the total, that discrepancy would be immediately obvious.
2. An error (even a major one) in an intermediate step may provide only a small (inconspicuous) error in the final outcome. Miscalculation on Chris's total will change the average by only one-third as much. If there were ten roommates, the difference would be smaller still. Thus, very reasonable sounding answers could slip by if the programmer only looks at the final answer.
3. Even if an answer is clearly wrong, a single large equation provides little hint about the exact source of the error. Suppose the results indicate that each person is owed money by the other two. Something is wrongbut what? Intermediate answers would again be more informative.

Intermediate values provide direct insight into the problem. Suppose the roommate spreadsheet also contained each student's college ID, as in Figure 14.1. Although the final values seem quite reasonable, both the individual totals and the average are obviously wrong. A quick error check for feasible final solutions may not detect the error. On the other hand, almost any quick approximation of the intermediate values reveals an error: the individual totals are way too high. In addition, the position of that error reduces the search space for the error: since something is wrong with each individual's total, the error must occur before that step. Thus a large problem (finding the error somewhere) is reduced to a smaller

problem (finding the error in the individual totals).

1 Expenses for the month of October						Totals
2 who	ID	purchases				
3 Chris	12345	\$5.00	\$20.00	\$13.00	\$92.00	\$12475.00
4 Pat	12379	\$437.00				\$12816.00
5 Joe	12387	\$37.00	\$37.00	\$37.00	\$37.00	\$12572.00
6						
7 average						\$12621.00
8						
9 Differences	Chris	\$146.00				
10	Pat	-\$195.00				
11	Joe	\$49.00				

Figure 14.1
Incorrect Expense Calculation

14.4

Testing and Verification

Men occasionally stumble over the truth, but most of them pick themselves up and hurry off as if nothing happened.

WINSTON CHURCHILL

The major mistake most beginners make is failing to test adequately (or at all). They may regard debugging as “fixing the mistakes that you notice.” But finding less obvious errors can be equally as important. Finding those errors requires systematic testing.

Errors only cause problems if they are not fixed. Unfortunately, the most significant reason errors do not get fixed is also a human foible: self-delusion. People want to see the correct answer and will think they see it much too early in the process.

The computer said so

is one of the most overused excuses in the world. Unfortunately, most new programmers do not look for subtler errors. It is all too easy to quit as soon as an answer “looks good.” Unfortunately not even a correct result for the initial set of input values necessarily means that an algorithm is completely correct, or will work for all inputs. The spreadsheet in Figure 14.2(a) appears to calculate total pay correctly. Unfortunately, it does not work for all values. Figure 14.2(b) shows the formula that produced the result. This function will leave the majority of workers quite unsatisfied, but the problem does not show up in the initial test.

14.4.1

Assume that your Program is in Error

You will usually be right. (And whenever you are wrong apparently, you will be right? If this seems paradoxical, see Box 14.1.) This is not a criticism of you, the programmer. It is simply a fact of life about programs: they are usually wrong. If you assume that there will be errors, you can build in the tools for detecting and

identifying errors. The assumption of error allows you once again to apply the scientific method. The assumption and question:

*I believe my program is incorrect;
How can I prove that?*

will prove a much better defense than any testing plan that starts with the assumption that the program is correct. It is often easy to demonstrate errors, but Gödel's theorem implies that you can never be 100% sure that a program is absolutely correct. Conclude that it works only when you give up on finding any more errors.

A	B	C	D	E
How many hours did you work?	<input type="text" value="2"/> hours			
What is your rate of pay?	<input type="text" value="\$2.00"/>			
Your check will be for:		\$4.00		
(a)				
How many hours did you work?	<input type="text" value="2"/> hours			
What is your rate of pay?	<input type="text" value="\$2.00"/>			
Your check will be for:		=D2+D4		
(b)				

Figure 14.2
A Poor Attempt at Calculating Pay

14.4.2

Plans

Have a *testing plan*. You must be able to test every program. That means that you must be able to select input data that tests all possible cases. The best time to think about a testing plan is when you write the program not when you realize that it does not work. Thinking about testing in advance will also help you to see flaws in your basic logic and specifications that you have overlooked. At each step ask the question:

How will I know if this step is correct?

Notice that even this simple question has two subparts:

What are the implications of incorrect results from this step? and How can I see those results?

14.4.3

Test Data

A plan helps the programmer anticipate errors and spend less time figuring out

2. For a description of the halting problem that is easily accessible by a non-computer scientist, see Walker, H. *The limits of computing*. Boston: Jones and Bartlett, 1994.

Box 14.1 “Impossible” Situations

The “if you do, you don’t” paradox above is actually a play on a famous logical conundrum, called Russell’s Paradox, after the British logician Bertrand Russell (1872-1970). Almost one century ago, the contemporary set theory of Gottlob Frege postulated that it was possible to create a set (collection of elements) composed of anything at all. Russell pondered the implications of this conjecture by asking about the existence of a set containing “exactly those elements that do not belong to any set.” By definition, no set belonging to another set is in this special set and every set in this set is in no set. An element would be in such a set if and only if it is not in any set—an apparent impossibility. A popularized version of Russell’s paradox describes “The Barber of Seville,” who it is said shaved every man in Seville who did not shave himself. This leaves the unanswered question “Who shaves the barber?”

Russell’s paradox (together with several variations) had a profound impact on twentieth-century mathematics and computer science. Perhaps the most famous of the variations, known as *Gödel’s Theorem* (after the German-American mathematician Kurt Gödel), implies that mathematics is not so universally expressive as was commonly thought. Essentially, *Gödel’s Theorem* states that any language capable of expressing the rules of arithmetic must necessarily be inconsistent. A second variant, known as the *halting problem*,² states that no computer program can determine if another program is actually an algorithm. The proof of that theorem depends on the construction of an algorithm that will halt (stop) if and only if it does not halt. If no program can determine if any proposed algorithm is truly an algorithm, it must be difficult indeed for a human to do so.

what went wrong. In the early stages of debugging, it is especially

important to have simple test data for which you know the expected result. When you test a multiplication step, do not simply test with the input 743×491 and look to see if the answer seems reasonable. Certainly such approximations will help find many of the most egregious errors, but not the subtler ones. Besides it will take you too long to check the results. Instead use data for which you know (or can easily find)

the expected result and compare. Use simple values and work up to more complex cases. Instead of 743×491 , use 1000×500 .

Approximation.

The first line of defense is approximation. Does the answer seem plausible? Devise and use rules of thumb (see Section 10.1, “Rules of thumb as relations”). Make sure that the answer is at least close to what you expect. Most people would be surprised if their income for the year were either \$1,000,000 or - \$5,000. A very quick approximation can rule out many of the most outrageous errors.

Cover All Variations.

Build a model or image of the circumstances under which you experience an error (and when you don’t). The description will help you understand possible sources of the error. Remember the test data you use (i.e., write it down).

Replicate the Results.

After you attempt to fix an error, you will want to replicate the situation with the same test data. Otherwise you will not know whether the change actually fixed the error or if the second test data simply did not generate an error. This means you must keep track of the test data that you use.

Select Test Values Carefully.

They should not only verify correct results but should also bring incorrect results to your attention. Avoid duplicate or overly simple values. These can lead to “coincidentally correct” answers. For example, since

$$2 + 2 = 2 \times 2 = 2^2$$

using 2 and 2 as test values may miss potential errors (as in the payroll problem).

A test should include examples of each appropriate type of input that might have any impact on the answer. In creating the plan, the programmer should ask, “What do I want the program to do in each of these cases?” Conversely, make sure that all cases are covered. Test data should include at least the following contrasts or conditions:

large and small values,

integer and noninteger values,

positive and negative numbers,

zero values,

limit cases (largest, smallest, etc.),

coincidence cases (e.g., what happens when two input values are the same),

values that force special cases, and

pairs of data with differing relationships to each other.

For each case, first ask, “Is it possible that the distinction could exist?” For example, there may be no need to test for integer values in a file that should contain a person’s name. But even in such a case, it might be worth knowing what will happen if the user inadvertently uses inappropriate data. Second, the programmer must consciously select data meeting the criteria.

Internal Relations

The error in Figure 14.2 would not show up with any of the following pairs of values:

Pay	Rate
2	2
3	1.5
1.5	3
1.25	5
5	1.25

because they all have the same duplicate relationships to each other:

$$pay + rate = rate \times pay$$

The programmer needs to select data carefully to ensure that sets of test input vary in their internal relationships.

Zero Values.

Many mathematical functions produce strange results given an input value of zero. In particular, division by zero is undefined. The programmer must make sure that the program will perform properly with an input value of zero. Alternatively, she must determine that zero is not a possible input value. The same is true of other null values such as the null string or the selection of zero cells.

Limit Cases.

Does the program perform adequately when extreme values are

used? For example, many programs have problems with very large numbers.

14.4.4

Check Points

Determine points in your program for which the results should be predictable. Use these points as a first line of defense against errors. Identify these points before attempting to run a program. Break spreadsheet equations into smaller ones. Many intermediate values are more easily predictable. Figure 14.3 shows two attempts to calculate the total cost of a clothing purchase (including tax). The correct formula may not be immediately apparent from the two bottom-line results in (a). On the other hand, the intermediate value, `tax`, in the first column seems exorbitantly high, suggesting that the other column may be better. Incidentally, notice in (b) that the coded differences are quite subtle.

total cost of a purchase		
Item	Price	
dress	\$34.72	
shoes	\$17.23	
belt	\$11.98	
total cost	\$63.93	\$63.93
tax	\$52.79	\$4.48
grand total	\$116.72	\$68.41

a purchase	
Price	
\$34.72	
\$17.23	
\$11.98	
=B4+B5+B6	=B4+B5+B6
=B4+B5+B6 *0.07	=C7*0.07
=B8+B7	=C7+C8

(a)

(b)

Figure 14.3
Intermediate Values as Error Indicators

Invariants.

An *invariant* is a relationship that holds throughout your program (or segment). For example, one column may contain the number of customers processed thus far. Use invariants as a first line of defense against errors. Identify these points before attempting to run a program. For example, in a double column of numbers the relationship between two adjacent numbers should be predictable.

14.4.5 *Seeking Help*

If you program long enough, there will come a point when you cannot find the source of an error. You will need to seek help. Course instructors are usually happy to help students, as are the consultants in computer rooms. The helper does not know of your problem or what you have done to find the solution. She will need information before she can help you, and you will get help faster if you can provide that information. To make the best use of outside help you should provide information on the most important aspects of your problem. The information she needs is essentially the same information that you would need yourself:

What was the symptom? Write down or memorize the error message before seeking help! An incorrectly described error message can lead the helper astray. Be sure you can answer the

question: “What makes you think there is a problem?”

What were the steps of the algorithm that apparently caused the error? That is, she probably needs to see the error itself. Know the answer to: “Exactly what were you attempting to do at that point in the program?”

What have you done to test the possible causes of the error?

14.5

Classes of Error

It often seems to the new programmer that almost anything can cause an error. In reality, a small number of mistakes account for the vast majority of student errors.

Thinking about the role of the software can lead to prevention and detection of errors. How or when the application detected the error provides important information about the nature of that error. Learn to recognize the differences; it will save much grief.

14.5.1

Syntax or Compilation Errors

These errors are detected by the application as it attempts to understand the code written by the programmer. Invariably, a syntax error message means that some aspect of the code is not a legal construct in the language of the application. The application indicates a syntax error only when it can find no valid way to interpret the instruction. Syntax errors are the most obvious of all errors: there will always be an error message. In fact, most spreadsheet applications will prevent you from continuing until you fix the problem, providing instead a message such as the one in Figure 14.4.4

The most common root cause of such errors is a careless or misunderstood use of a command, such as

a misspelled identifier or label (e.g., 3D instead of D3)

an incorrect specification of operation and operand (e.g., d3+ +d4 instead of d3+d4)

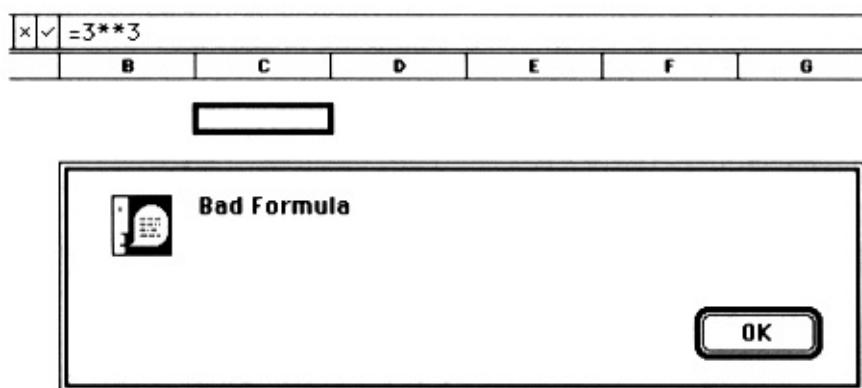


Figure 14.4
An Application-Detected Syntax Error

3. The word *compilation* is not usually used in the context of a spreadsheet or database. It refers to the task of interpreting the instructions that the user entered. Most end-user applications now use the term *syntax errors*, but *compilation* is common throughout computer science subject areas, particularly for errors in higher-level languages.
4. Note that in some languages, the expression used in the figure is actually legal and means 33. Errors such as this illustrate how easy it is for a programmer to make an error when using an unfamiliar application.

- an incorrect number of arguments to a built-in function (e.g., `sqrt(3, 4)` instead of `sqrt(34)`)
- an incorrect parentheses [e.g., `(3 + 4)` instead of `(3 + 4)`].

Corrective Action.

The error must be contained at least partly within the current statement. There is no need to look at the algorithm itself, other statements, or data values, or to check your disk for errors. Look for obvious errors. Check each of the four common problems above. If necessary, check the syntax in the manual or online help. Because most commands have few syntax rules, a small set of precautions will prevent or identify most errors. Remember that the error message describes a problem with the application's attempt to interpret the instruction not with the actual error that the programmer made. The wording, therefore, may be misleading. Instead of looking for errors that match the wording of the message exactly; focus on errors that might be able to cause the error message.

14.5.2

Run-Time Errors

Run-time errors are not detected until the program actually performs a computation. In a spreadsheet, this means when the application attempts to calculate the functional value in a cell. They occur when a legal expression results in an operation that cannot be performed using the given data. The statement must be syntactically correct, but have a significant problem when applied to the given data. Typical run-time errors include division by zero or nonnumeric values used as numbers. Most spreadsheets indicate run-time errors with very cryptic messages such as those in Figure 14.5. Notice that an empty cell (A1) is treated as zero, creating the divide-by-zero error in the first row.

Corrective Action.

Check that the data cells referenced by the cell in error are indeed the ones you intended (e.g., was it really A1 that you wanted for a denominator, or was it B1?). Check for typos (e.g., 3/A1 when you meant 3 + A1). Check the definitions of built-in functions for misunderstood parameters (does month want the name of a month or a full date?). There is no need to look beyond the cell where the error is displayed or any cell that it refers to explicitly.

	A	B	C	D	
1	=3/A1		divide by zero		B
2	a	=A2+2	adding a non-numeric value		#DIV/0!
3	-2	=SQRT(A3)	square root of a negative number		#VALUE!
4	1234567890	=A4	value will not fit in cell		#NUM!
					#####

Figure 14.5
Some Run-Time Errors

Calculation Versus Display Errors.

The large value in the last row of Figure 14.5 is not really an error in the usual sense. The program calculated the value correctly, but could not fit the result in the cell. For historical reasons, most spreadsheets will attempt to identify actual calculation errors, but use *fill characters* such as "#" or "*" to indicate output problems. Such display errors can be fixed by making the cell wider.

Cascaded Errors.

A single error can cause multiple errors or at least multiple error messages. That is, the error can cascade down a program as in the Figure 14.6, which seems to suggest that there are two divide-by-zero errors. Part b shows that in fact the second error is caused by the first. In general, find the earliest (which will usually be the topmost or leftmost in well-structured programs) error and fix it first. This single correction, or *patch*, may remove more than the single error message.

14.5.3

Machine Error

Machine errors are not errors caused by the machine as the name seems to suggest. Rather, the machine (or the application) stops because an operation cannot be performed. The error was caused by a catastrophic problem that could not be known until run time. Machine errors will usually cause major observable results. They seldom produce an incorrect answer and continue to run. The symptom of a machine error is usually that the machine stops running entirely. Most modern applications protect the user from such errors. That is actually what the run-time error message says: the application intercepted a major problem. In fact, protecting the user is one of the major tasks of the application. But machine errors do occur occasionally. They are not usually the user's fault. In fact, the manufacturer of the program would probably want to know if

you encounter a genuine machine error caused by your program.

Corrective Action.

Fortunately for students, most instructors will not hold students responsible for problems in assignments when *bona fide* machine errors occur (but do note that these errors are extremely rare; you will seldom be able to use the excuse). Unfortunately, an excuse does not mean your program will work. And if you need the answers for real world problems, you must find an alternative

#DIV/0!	=2/0
#DIV/0!	=1+B8

Figure 14.6
A Cascaded Error

solution. This may mean a new algorithm or a new representation for the single line causing the error.

Check to see if you have a genuine error or a random accident by trying to reproduce the results (you did save a backup copy, I assume). If the error occurs regularly or predictably, there is a bug in the application; seek help from a consultant. If you cannot reproduce the error, forget about it (and be thankful for your backup).

System or File Error.

Occasionally, a program cannot complete a major operation on an entire file. For example, it may not be able to open a document, or the application itself cannot even start. Errors such as this are not normally caused by anything in the document. The disk may be bad or there may not be enough memory available. Do not waste time inspecting the logic of your program to correct errors that occur when opening the program or the document. Instead, read the message carefully. It is your best starting point. In the case of a bad disk (or file), attempt to copy the file and then read the new one. If there is not enough memory, quit from another application.

14.5.4

Logical Errors

Logical errors are the most diabolical class of error. They produce no error message and the program generally acts as if it were correct: values or results appear at all appropriate locations. The program looks finebut the answers are wrong. Unfortunately this is a very common situation. Just as unfortunately, new programmers often believe their program is correct when it has no syntax or run-time errors. Logical errors can only be detected by comparison of the actual and expected output. They, therefore, give programmers the most difficulty. Most of the techniques in this chapter are actually aimed at detecting and preventing logical errors, such as

the payroll problem. There is no detectable error, but the total pay would be very disappointing to most employees.

Corrective and Detective Action.

Fixing logical errors is really what debugging is all about. And detecting them is the purpose of testing. All of the other classes are “self-detecting.” Follow the suggestions of this chapter.

14.6

Misconceptions About Good Programming

For various reasons, a number of misconceptions about programming have crept into the folklore. These misconceptions can be doubly diabolical. First, they seem to make sense or to have a basic appeal. Second, they are occasionally presented as good practice, which seems to promote them from an idea to a “recommended idea.” Do not fall victim to these misconceptions in programmer’s clothing:

Short is beautiful. Many new programmers believe that the shorter a program or description is, the better: a spreadsheet with a few less cells, a database with a

few less entries, an outline with fewer levels. There is nothing wrong with short or elegant solutions. But short is not a goal in itself. Never make a program shorter when a few extra steps would make it clearer.

Faster is good. While it is true that we all prefer to get an answer as quickly as possible, there is little an applications programmer can do to speed up a computer's computation. Careful use of the tool may speed up an algorithm by a fraction of a second. But the fraction of a second is significantly smaller than the minutes or hours the human may need to develop the improved version.

Clever is wonderful. Clever is not a crime. But it can be very confusing or misleading. Never substitute cleverness for clarity. Given the choice of being tricky or clear, always be clear. The millisecond you save will never be noticed; the extra hour of confusion will be. If your program is to be run thousands of times, you can easily optimize it later if the original code is well written.

Get it running quickly. Avoid the temptation to get a program up and running as quickly as possible. The time spent debugging is usually greater than the time spent with the original program. Any steps you can take in the design process to reduce errors will be time well spent.

It's not worth the effort! When confronted with the steps needed to assure good results, some new computer users conclude that using a computer just is not worth the time. They conclude that since they have to test all of the calculations themselves, there is no point in writing the algorithm. But recall that the reasons for creating algorithms in the first place were: precision, reuse, debugging, and sharing the labor (see Section 12.1). Reuse is a particularly important reason: once defined on test data, you can use the algorithm over and over without significant rechecking. In addition, the following chapters include methods for reliably calculating many more values than you actually test.

The designer who falls into the trap of placing such criteria before error prevention will pay with many hours of error correction.

14.6.1

So your Program is More Efficient, is it?

Efficiency is not the primary goal of programming. However, the concept of efficiency cannot be completely ignored. *Efficiency* is actually a technical term in computer science. Its most common measures are:

the quantity of memory used, and

the amount of time required to run a program.

Usually these are stated as a function of the quantity of input. One criterion that is not a measure of efficiency is the size of the code. Also computer scientists are not interested in small changes in these measures. Suppose you have a machine that executes one million instructions per second and can store one million characters in memory (both very modest assumptions). A program that executed 100 fewer instructions than another would run 1 ten-thousandth of a second faster. A program that executed 10,000 fewer instructions would only run 1 one-hundredth of a

second faster. In practice, code that is only executed once effectively adds nothing to the total computation time. Put another way, no one could write so much code that the speed would matter if it were only executed once.

Similarly, fewer cells provide no significant savings in either space or time. Any perceived savings is worth nothing compared to the loss in readability.

14.7

Summary

Unfortunately, errors are ubiquitous. Systematic debugging and anti-bugging will reduce the amount of time spent dealing with these errors. Learn to identify the sources of error (incorrect algorithms, incorrect representation, incorrect interpretation of results, and incorrect data), and the points of detection (syntax or compilation errors, run-time errors, machine errors, and logic errors). Take careful action to correct the errors: test and debug, anti-bug, and finally, seek help.

Important Ideas

syntax	run-time	machine
error	error	error
logic	debug	anti-bug
error		
testing	verification	

15

Iteration: Replicated Structures

Men get opinions as boys learn to spell, By reiteration chiefly.
ELIZABETH BARRETT BROWNING (*Aurora Leigh, Book 6*)

Pointers

Gateway Iteration

Lab:

Lab Unit 9:

Manual: Iteration

15.0

Overview

Iterate means “to do again.” As stated repeatedly in this text, reuse or performing again is a major advantage for creating general algorithms. Iteration generally refers more specifically to actions or structures repeated within a single program. Every database seems to have many items defined in the same way. Most functions are defined in the same way for each independent variable. By this point many students have run into situations that begged for a method or command meaning, “Now do the same thing again 10 more times.” Iteration is that method.

15.1

Iterative Data Structures

Algorithms for calculation seldom involve only isolated values. Blocks of data with very similar definitions seem much more common. By definition, the elements of relations have parallel

structure, as do the records of databases. Often this similarity takes the form of a repeated function or a cell or line definition. For example, Table 9.3 showed a list of ten numbers and their squares.

1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81
10	100

Each line is defined in exactly the same way:

An integer on the left, and its square on the right.

Similarly, the housemate expense calculation in Chapter 15 contains several examples of repeated structures:

The purchases of one roommate are a set of identically defined numbers.

For each roommate, the total spent is calculated in exactly the same way:

Add up all of the purchases for that person.

The amount owed by each person is found in exactly the same way.

(The calculations are repeated in Figure 15.1.)

15.1.1

Table Organizations for Data

A block of data that can be described with an iterative definition is variously called a *table*, a *matrix*, or an *array* of data. In fact, you will recall (see Chapter 8) that the repetitive nature of data was essential to the organization of a database. Clearly, the neat formatting of columns and rows of a table makes reading the resulting collection of data much easier. In the roommate example

(Example 13.1),

1	Expenses for the month of October						
2	who	purchases					Totals
3	Chris	\$5.00	\$20.00	\$13.00	\$92.00		=B3+C3+D3+E3+F3
4	Pat	\$437.00					=B4+C4+D4+E4+F4
5	Joe	\$37.00	\$37.00	\$37.00	\$37.00	\$37.00	=B5+C5+D5+E5+F5
6							
7	average						=(G3+G4+G5)/3
8							
9	Differences	Chris		=G7-G3			
10		Pat		=G7-G4			
11		Joe		=G7-G5			

Figure 15.1
Roommate Calculations (Repeated from Figure 13.1)

the position of each cellrow and columnhelped define the meaning of the data in that cell:

Each row represents one person.

Column positions show the order of purchases by an individual (first purchase at the left (column B) and last purchase at the right (column F)).

Generally, the position of a data itemboth row and columnsshould convey meaning. The example of the list of ten numbers and their squares contains pairs of related numbers. Two numbers in the same row are related. Numbers on the left have one interpretationthe number to be squared, the raw data, or independent variables. Values on the right are the dependent variables: the actual squares. Roughly, in this example the row is similar to a record in a database and a column is similar to a field. In fact, even the definition of relation (see Section 8.1) suggests that each row of a relation can generally be constructed in the same way. Any relation or table containing one column defined on the basis of other columns is a candidate for definition by iteration.

15.2 Control Structures

The structure for any algorithm involving a structured set of dataeither as data or as derived valueswill reflect that parallel structure. This structure provides a valuable aid for creating the algorithm. In the squares example, the relationships are the same for each row. The cells in the table are described as:

1	12
2	22
3	32
...	...
10	102

The mathematical expressions in each position describe how that position is calculated. The values in each row of the table are calculated in exactly the same way:

For each row:

the number on the right is the square of the number on the left.

Generally, algorithms for computing tables often take the form:

For each element (or row):

do some action.

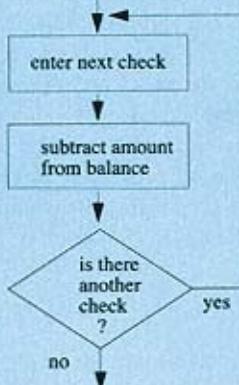
where *some action* is a specific repeated step or group of steps in an algorithm. Stated from the opposite perspective: any algorithm with a segment of the form:

For each of several variables:

calculate its value through some action.

Box 15.1 Iteration and Loops

You may have heard computer users refer to loops or infinite loops. A loop is nothing more than iteration. The name comes from the appearance of an iterative algorithm in a once-common graphical representation called a *flowchart*. A simple algorithm might look like:



The structure clearly looks like a loop. Recall that the definition of *algorithm* required that the algorithm be finite. Picturing iteration as a loop helps demonstrate how a program could fail the finiteness test and therefore fail to be an algorithm. An infinite loop is any loop that never exits, either because there is no exit question, or because the question can never be false. The most famous infinite loop appeared for many years on—of all places—shampoo bottles:



The problem with this algorithm is it contains no test for determining whether or not more repetitions are needed. That is “For each” is not well defined. For more on loops and infinite loops see *Chapter 25: Control Structures Revisited*.

can be expanded to several nearly identical steps, each describing the calculation for one value of the table. In the original development of the roommate calculations:

For each roommate:

find the total of all the items to its left.

was rewritten in an expanded form:

*Find the total paid by Chris (items to the left),
 by adding up Chris's purchases.*

*Find the total paid by Pat (items to the left),
 by adding up Pat's purchases.*

*Find the total paid by Joe (items to the left),
 by adding up Joe's purchases.*

The value of *total* in each row of the table is calculated in the same way. Either description creates the same result. But the former is clearly easier to write. The use of such statements is called *iteration*. Iteration is referred to as a *control structure* because it alters or controls the order of the steps in an algorithm. The normal interpretation for an algorithm is:

Do each step sequentially.

The iterative control structure says:

Steps should be repeated.

The first and most obvious advantage can be seen in the algorithm's definition: the written algorithm is shorter, or has fewer distinct steps. More importantly, the shorter algorithm is easier to write and easier to understand. It will also be easier to translate into a spreadsheet representation. Finally, algorithms which exploit iteration provide fewer opportunities for error.

15.2.1

Iteration and Spreadsheets

Because iteration is such a common and useful tool for representing algorithms, spreadsheets provide a tool for saying:

Do the same thing on the next line(s).

Obviously the programmer must define the relation one time. But once defined, a cell definition may be iterated into the following cells, creating a vertically defined table. For example, in Figure 15.2 on page 258, (really a repetition of Figure 9.4), the definition of cell C4 is “ $=B4^2$ ”, which can be paraphrased as “the square of the cell immediately to the left.” The definition of each cell below C4 is exactly the same. The programmer iterated the definition into the cells below, telling the spreadsheet “Fill the cells down from C4 with exactly the same definition.” This

B	C
table of squares	
number	square
1	=B4^2
2	=B5^2
3	=B6^2
4	=B7^2
5	=B8^2
6	=B9^2
7	=B10^2
8	=B11^2
9	=B12^2
10	=B13^2

Figure 15.2
Calculating Squares

action is so common that all spreadsheets provide a shortcut command to iterate the definition of one cell into each of the cells below.¹ A horizontal table can be defined similarly, with a cell definition iterated across the cells to the right.² Notice that iteration saves the programmers not the computer time and effort.³

15.2.2

Iteration is Ubiquitous

Iteration is both common and useful. Examples of data that can be defined iteratively can be found everywhere. The roommate problem included at least two relations that could be defined iteratively: the calculation of each individual roommate's total purchases and the final calculation of the difference between those totals and the average.

1. The name for this command in spreadsheets is almost universally: `Fill down` because it copies or fills the same definition into each cell below the original. The programmer could create the identical algorithm by entering the definition of each cell explicitly. In a completed spreadsheet, descriptions entered using the `Fill down` command appear exactly as cells filled in manually. The only difference is in the actions taken by the programmer to put the descriptions into the cells: automatically with `Fill down` or manually.

2. The name for this command in spreadsheets is also almost universally:
`Fill right`.
3. In other applications, iteration will actually have a different appearance, going by colloquial names such as `For`, `While`, `Do`, or `For next`, but the principle is the same.

Example 15.1 Professor Chin wants to automate her records for grading purposes. Her announced grading criteria state that the final grade will be based on:

homework	20%
midterm	20%
labs	25%
final	35%

For each student, she must first calculate two partial grades, and then calculate a total grade from those, as in Figure 15.3. Since every student's grade is calculated in exactly the same way, she first defines the grade for a single student (Billy) and then copies the definition into the lines for all other students. Cell P3 contains the total grade for a student named Billy. (Cells G3, L3, M3, and N3 contain Billy's total homework, total lab, midterm, and final exam grades.) The rows below Billy will contain parallel descriptions for the other students. Such repeated definitions are extremely common, for example:

- Your monthly budget of college expenses
- Checkbook records
- Payments on a loan
- Collections of scientific data
- An address book
- Your grade transcript
- An instructor's class list
- An index to another book
- A directory of computer files
- A price list (especially one containing discounts on each item)
- Customer records for a small business
- Any enumerated table (such as the list of squares).

	A	B	C	D	E	F	G	H	I	J	K	L
1	name	homeworks							labs			
2		#1	#2					total				
3	Billy						=SUM(I					
	L	M	N	O		P						
		MT	final		total							
	=G2											
		=SUM(L					=0.2*N3+0.2*L3+0.25*M3+0.35*N3					

Figure 15.3
One Student's Grade Calculation

Exercises

- 15.1 Add five more items to the above list.
- 15.2 If you have not done so yet, work through the Gateway lab on iteration.
- 15.3 List your grades for one semester on a line, with your GPA at the right end of the line. Iterate the definition down for all the semesters you have been (will be) in college.
- 15.4 Create a budget page. Each line should represent one month and contain the name of the month, several categories of expense, and a total monthly expenditure.
- 15.5 Repeat Exercise 9.14. This time use Fill down to extend it further.
- 15.6 Repeat Exercise 10.4. This time use Fill down to extend it further.

15.3

Iteration as an Algorithmic Tool

Every iterative data structure suggests an iterative algorithm. Iteration is clearly an excellent tool for creating larger algorithms from smaller ones, whenever the data is repetitious in structure. They are also useful for data collections with not-so-simple structure.

Example 15.2 Order Form. The BigByte Computer Company uses an order form containing several lines, one line for each item in a purchase. Each line has several columns for:

- product name
- product number
- quantity

price

discount (percentage, different for each customer and product)

net price = $(1 - \text{discount}) \times \text{price}$

total cost = net price \times quantity

Most of the items are entered individually for each row. But the definitions of net price and total cost are functions defined identically for each row. To create the table, define the first row and copy those two columns down through the remaining rows:

	A	B	C	D	E	F	G	H
1	BiG Byte Computers, Incorporated							
2	Order sheet							
3	Item Product							
4	number	name	number	quantity	price	discount	net price	total cost
5	1	Granny Smith	1234	2	\$999.99	7%	\$929.99	\$1859.98
6	2							
7	3							
8	4							

The values in the other columns are not affected. In fact, this corresponds to the definition of function: some values (dependent variables) in each row should be predictable from the others (independent variables).

15.3.1

Toy Problems and Algorithm Development

Problems requiring the calculation of many values may sometimes be intimidating. Iteration reduces a large problem to a small problem through the simple credo:

*If you can do it once,
you can do it many times.*

Certainly this is an instance of the bottom-up coding technique discussed in Section 13.4. Building one line is a simple and very low-level task. Iteration builds a higher-level construct or algorithm from the low-level one.

A *toy problem* is any problem built in the image of a real problem only smaller. In the case of an iterative structure, a single cell of that structure contains the essence of the entire structure. One good way to start building a large structure is by building the corresponding toy problem. Create a single cell definition. Test that cell. When you are confident that it is correct, then (and only then) fill in the remaining cells. Until you are confident in your solution you need only work on the much smaller (toy) problem. The small problem almost always seems easier than the full problem.

15.3.2

Iteration and "What-if" Problems

Iteration is a natural tool for “what-if” problems. It allows the user to perform any number of identical calculations. For example if you needed to compare the total costs of several proposed investments, you could build a table such as in Figure 15.4. Notice that this

relation includes two calculations per line and that these must be calculated in the correct order: the same order for each instance of the relation. As you have seen (e.g., Section 12.4), spreadsheets are ideal tools for working with possible scenarios or “what-if” problems. Iteration can help with “what-if”

2	item	cost	tax rate	tax	total
3	fur coat	\$1000.00	7.0%	=B3*C3	=B3+D3
4	new computer	\$2000.00	7.0%	=B4*C4	=B4+D4
5	old car	\$1800.00	7.0%	=B5*C5	=B5+D5

2	item	cost	tax rate	tax	total
3	fur coat	\$1000.00	7.0%	\$70.00	\$1070.00
4	new computer	\$2000.00	7.0%	\$140.00	\$2140.00
5	old car	\$1800.00	7.0%	\$126.00	\$1926.00

Figure 15.4
Iteration as a Comparison Tool

investigations in two ways, which might be called *comparison shopping* and *how many times?*

Comparison Shopping.

The power of a spreadsheet for “what-if” problems comes from the ease with which you can change the scenario by entering new initial data. For many problems, the question is not so much “what-if,” but “which is best,” or “what are all the viable possibilities?” In such cases you want to be able to see multiple scenarios at the same time. For example, suppose after one semester of college, you (or, more likely, a friend of yours) are on academic probation (assume the cut off is 2.0). In your first semester you earned a grade point average of 1.85 for 15 credits. Clearly you need to get a 2.15 for this semester (assuming you again take 15 credits). But what grades do you need exactly? What if you do not do well in history? How much worse is it to earn a bad grade in computer science? A “what-if” scenario for solving this problem can be iterated to create a table of scenarios⁴ as in Figure 15.5. One can see immediately that all “C”’s with a “B” in either history or computer science (but not art) is sufficient. On the other hand a “D” in history can be balanced by two “C”’s, but a “D” in computer science cannot. (Moral: do well in computer science.⁵) (Exercise 15.7 asks you to work out the details.)

course	credits	Possible scenarios					
		one B . . .	one D . . .	one B . . .	one D . . .	one B . . .	one D . . .
history	3	2	2	3	2	2	1
computer science	4	2	3	2	2	1	2
math	3	2	2	2	2	3	3
art	2	2	2	2	3	2	2
sociology	3	2	2	2	2	3	3
quality points (grade times credits)							
history		6	6	9	6	6	3
computer science		8	12	8	8	4	8
math		6	6	6	6	9	9
art		4	4	4	6	4	4
sociology		6	6	6	6	9	9
Grade point average		2.00	2.27	2.20	2.13	2.13	2.20

Figure 15.5
What Will It Take for Me to Stay in School?

4. It would of course be nicer to enter A, B, C, than 4, 3, 2. See Chapter 17 for a way to make this improvement.
5. Actually, it's "do well in the courses which give more credit," but I like the sound of my answer.

Exercises

15.7 Create a spreadsheet equivalent to the grade point spreadsheet above, but customized to your own needs. Notice that iteration can be used to define the entire quality point table as well as the averages.

15.8 Improve the spreadsheet of Exercise 15.7 so that it works even if you have completed more than one semester or if you take a varying number of credits each semester.

15.3.3

Iteration and Nonnumeric Series

Previous examples in this chapter used numeric functions. Many may be defined iteratively. The initials and date examples of Section 11.2 iterate across an entire collection of records. For example, if the three column names of politicians, the function

```
D2 = concat(substring(A2, 1, 1), substring(B2, 1, 1))
```

could be iterated down column (D) to create the corresponding list of initials. 15.6.6 Or, if BigByte (Example 15.2) estimated that shipping takes 7 days, an order form contained the date an item was actually shipped, the estimator generated by iterating a calculation of the date a week later:

```
J5 = DATE(YEAR(15), MONTH(15), DAY(15) + 7)
```

across all rows as in Figure 15.7. This example provides a good way to see both iteration over defined tables and the algorithmic process. Each row contains the three individual parts of the shipping date: day, month, and year. The formula `DATE` is used to put the pieces together to create a complete date. It builds the date from its constituent parts just as `concatenate` builds a string from smaller components. “build a date,

2	Franklin	Delano	Roosevelt	=CONCAT(LEFT(A2,1),LEFT(B2,1),LEFT(C2,1))
3	John	Fitzgerald	Kennedy	=CONCAT(LEFT(A3,1),LEFT(B3,1),LEFT(C3,1))
4	Richard	Milhouse	Nixon	=CONCAT(LEFT(A4,1),LEFT(B4,1),LEFT(C4,1))
5	James	Earl	Carter	=CONCAT(LEFT(A5,1),LEFT(B5,1),LEFT(C5,1))
6	William	Jefferson	Clinton	=CONCAT(LEFT(A6,1),LEFT(B6,1),LEFT(C6,1))

Figure 15.6
Iterating a Text Function

6. As noted earlier the exact name and form of substring commands varies by applications. This particular example uses *ClarisWork*'s `left` function, which returns the number of characters from the left-hand end of a string.
7. Of course for some dates this would be in the next month. Ignore that problem for now. See Exercise 17.12 for a method for changing the “month” in those cases.

shipping date	delivery date
Sep 6, 1995	=DATE(YEAR(15),MONTH(15),DAY(15)+7)
Oct 3, 1995	=DATE(YEAR(16),MONTH(16),DAY(16)+7)

shipping date	delivery date
Sep 6, 1995	Sep 13, 1995
Oct 3, 1995	Oct 10, 1995

Figure 15.7
Iterating a Date

using the year and month from a cell to the left, but calculate the day as ‘seven more than that date’.” In general, any time a defined field has the same definition for several records, that field is a candidate for iteration.

Exercises

15.9 In some spreadsheet applications, the built-in function DayName returns the name of the day. For example, Dayname (now ()) returns Tuesday . (Most applications will have an equivalent function. Use your help facility to find it.) Build a table showing the names of the days for the next month.

15.10 Build a table showing the age you will be on your birthday every year for the rest of your life. It should start this year. Each line should contain the date of your birthday that year (e.g., 1/2/94) and the age you will be on that day.

15.3.4

Double Iteration

Example 15.2 showed that the items that are built in one row can themselves be complicated structures. In fact, the step to be iterated can itself be any algorithmic structure. Some tables have definitions that are iterative in two different directions. The multiplication table in Figure 15.88 is an iteratively defined structure in both the vertical and horizontal directions. One algorithm for creating such a table is:

For each row:

build each column in that row.

But within a row, the columns are defined by:

For each column:

calculate the value for the cell in that column.

8. The actual implementation of this example in some spreadsheets applications will require one addition concept: relative addressing, described in Section 16.3.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Multiplication table												
2		1	2	3	4	5	6	7	8	9	10	11	12
3	1	1	2	3	4	5	6	7	8	9	10	11	12
4	2	2	4	6	8	10	12	14	16	18	20	22	24
5	3	3	6	9	12	15	18	21	24	27	30	33	36
6	4	4	8	12	16	20	24	28	32	36	40	44	48
7	5	5	10	15	20	25	30	35	40	45	50	55	60
8	6	6	12	18	24	30	36	42	48	54	60	66	72
9	7	7	14	21	28	35	42	49	56	63	70	77	84
10	8	8	16	24	32	40	48	56	64	72	80	88	96
11	9	9	18	27	36	45	54	63	72	81	90	99	108
12	10	10	20	30	40	50	60	70	80	90	100	110	120
13	11	11	22	33	44	55	66	77	88	99	110	121	132
14	12	12	24	36	48	60	72	84	96	108	120	132	144

Figure 15.8
Multiplication Table

Combining the two algorithms gives:

For each row:

*For each column: (in that row)
calculate the value for the cell in that column.*

Calculating the value for an individual cell is easy. It is just:

this cell = column number \times row number.

Inserting this (toy) definition into the algorithm yields:

For each row:

*For each column: (in that row)
calculate the value for the cell:
this cell = column number \times row number.*

The single calculation in each cell is both obvious and easy to test. Recognizing the double iterative nature of this data structure reduces a complicated problem to an easy one. Not surprisingly, the corresponding spreadsheet can be built with one `Fill down` and one `Fill right`.

15.4

Programming Techniques and Iteration

15.4.1

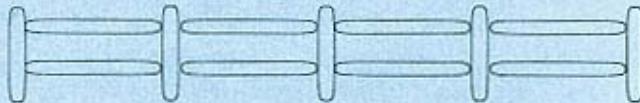
Implications of Iteration on the Design of Algorithms

Think about iterations during the algorithm development stage.
Potential iterations are usually obvious:

Box 15.2 Birthdays, Fences, and Counting from Zero

You have seen several instances of counting starting with zero. In iteration, you should always consider starting with zero. Let's see why that is. Consider a young boy anxious for his upcoming birthday party on Saturday. Today is Wednesday. The boy attempts to figure out how long he must wait by counting on his fingers. He counts: "Today is Wednesday (one), tomorrow is Thursday (two); the next day is Friday (three) and then the party on Saturday (four). But he really has only three days to wait. If he had started counting with zero he would not have made the mistake.

These errors are so common in computer science they are called *fence post errors*. Fence post errors? Why fence post errors? Consider the following problem faced by another child. To occupy her time while riding in the country, she decided to count the number of fence rails. Easy—just count the number of sections and multiply by two. But the rails were hard to see as she sped by so she counted the posts. After all, there is one post per segment, so that should be equivalent to counting the segments. Unfortunately there is one more post than there are segments. Any error of this general form is now called a fence post error.



As you develop an algorithm, look for definitions and actions that are the same or similar. They can be written as

For each whatever-it-is:

perform a calculation in terms of the related data.

Often iterated segments are “flagged” by phrases such as “for each,” “for every,” or “ten times.” Algorithms (or segments of algorithms) tend to take a general and common form. This form soon becomes an old friend, a technique providing comfort through its familiarity.

Any iterated segment expands into a series of steps, each of the same form. That provides an automatic step in your design of the final algorithm.

Finally, when you wish to represent such segments in a spreadsheet, use the iteration command (e.g., Fill down or Fill right).

15.4.2

Scope and Abstraction

The previous section introduced a subtle visual distinction. Prior iteration examples had two lines, a control structure:

For each row:

and an indented body line

perform some action

The double iteration examples contained lines that were indented twice. The context of

For each row:

For each column: (in that row)

calculate the value for the cell in that column.

made the meaning clear: the first line implies “the following lines are to be repeated.” The second line implies “the following line is to be repeated.” In each case, it is the lines indented relative to the control statement that are repeated. That is, indentation is a tool for representing the *scope* of the iteration control structure. As with arithmetic operators, it is essential to provide an unambiguous representation for the scope of control structures. Indentation plays the same role for algorithms as parentheses do for arithmetic expressions: they *delimit*, or mark the limits of, the scope. In the case of algorithmic representations, the scope of each control structure is normally: all indented lines immediately following the structure. As you build more complex algorithms, the concept of

scope will become increasingly useful.

15.4.3

Iteration and Debugging

Iteration helps explain the quandary posed earlier (Section 14.6):

Why bother to compute if you must check every value?

There is no need to test every instance of an iterated relation. Since each instance is defined in exactly the same way, it is really only necessary to test critical examples. Usually this includes:

the first instance

the last instance

at least one sample instance in the middle

any special cases (e.g., for dates, the beginning and end of a month)

This means that very large iterated calculations may require relatively little error checking. For a long invoice spreadsheet, even with hundreds of purchases, the programmer need only check perhaps four or five lines individual calculations.

15.4.4 Execution

Unless there is a very compelling reason, no loop should ever be partially executed. Do not insert noniterated items into an iterated structure. The visual structure of the spreadsheet should reflect the structure of the algorithm. Once a column is iterated, avoid changing the definition in the middle of that column. Fill down implies that all entries are the same. Avoid “short-circuiting” the process. If a change is necessary, document it. Figure 15.9 shows an iteration that appears to extend further than it really does.

	A	B	C	D	E	F
1	a dangerous practice					
2	1	2 =A2+B2				
3	2	4 =A3+B3				
4	3	6 =A4+B4				
5	4	8 =A5*B5	<- this shift will confuse the user			
6	5	10 =A6*B6				

Figure 15.9
A Confusing Broken Iteration

15.4.5 Documentation

Iteration presents a special situation for documentation. For each element of an iterated series, the relationship between adjacent cells in each line is normally the same. Each line or record should have the same description or “one description fits all.” That is, documentation of the form

*Each of these cells
represents the same function applied to specific other cells.*

should be possible. In addition, a given line can normally be

described in terms of its position within the block of code: the initial relation, the final or last result, or an intermediate result.

Exercises

15.11 Build the “addition table” analogous to the multiplication table of Figure 15.8.

15.12 Build a table of powers: $2^2, 2^3, 2^4, \dots, 3^2, 3^3, \dots$

(table continued on next page)

(table continued from previous page)

15.13 Comment on the number of rows and columns needed in the previous two exercises to define the matrices. Relate the total number of rows and columns to the number of values defined.

15.14 Repeat Exercise 11.19. This time use `Fill right` to extend it to a tool for translating an entire sentence. (Assume that the user puts one English word into each cell in a row.)

15.5

Summary

Iteration is a powerful tool for constructing both data and algorithms in which component parts are repeated. Iteration allows a programmer to construct a complicated algorithm by replicating individual sections of commands of an algorithm. Implicitly they create a tool for program development, called toy problems.

Spreadsheets explicitly recognize the iterative control structure through the `Fill down` and `Fill right` commands.

Important Ideas

iteration loop fence post
 error

fill toy table
 problem

matrix array control
 structure

flow
chart

16

Iteration: Replicated Algorithms

Iteration, like friction, is likely to generate heat instead of progress.

GEORGE ELIOT

Pointers

Gateway Iteration

Lab:

Lab Unit 9:

Manual: Iteration

16.0

Overview

This chapter could be titled: “looking at iteration again.” Iteration is a very powerful technique. It occurs in virtually every subcontext of computer science. This chapter reiterates some of the ideas of iteration, casting them in new contexts and revealing new problems and new solutions. In particular, iteration need not be restricted to self-contained functions or database records. Well used, iteration will indeed create “progress, not heat.”

16.1

Sequences and Recursively Defined Lists

16.1.1

Simple Sequences

Tables commonly require numbered lines or columns. Unfortunately, although spreadsheet applications automatically provide row numbers, these often do not reflect the structure of the

data. Because most tables have header information at the top, the item number probably will not be identical to the spreadsheet row number (e.g., item 1 was in row 5 on the BigByte order form in Example 15.2). In a check register, the number of each row should be the check number. In the table of squares at the beginning of the previous chapter, the rows are effectively numbered with the value to be squared. Programmers often want to provide their own numbering for the rows or columns of a section of the spreadsheet. While it is

possible to enter each row number individually, numbering seems to be an ideal candidate for iteration.

In a simple arithmetic series 1, 2, 3, . . . each number is exactly one more than the preceding number. This suggests an algorithm for filling in row numbers:

Put 1 into the first row number.

For all remaining rows:

put one more than the previous row number into the current row.

That is, after placing 1 into the first row number, place one more than that (2) into the next row number, and one more than that (3) into the following row number and so on. Writing this a bit more formally gives:

Example 16.1

Put 1 into the first row number.

For all remaining rows:

row number = (number of row above) + 1.

In a spreadsheet, this is easy:

1	item number
2	1
3	=A2+1
4	=A3+1
5	=A4+1
6	=A5+1

item number
1
2
3
4
5

Any simple series, such as:

2, 4, 6, 8, . . .

3, 6, 9, . . .

0.5, 1.0, 1.5, 2.0, . . .

1, 2, 4, 8, . . .

10, 9, 8, . . . 1

1, 0.5, 0.25, 0.125, . . .

can be similarly defined: enter the first element explicitly, but define every other element in terms of its predecessor, for example:

9	2
10	=A9+2
11	=A10+2
12	=A11+2
13	=A12+2
14	=A13+2

2
4
6
8
10
12

16.1.2

Nonnumeric Series

Many nonarithmetic series can also be defined iteratively. Consider the details of your car payments. Perhaps a payment is due on the 7th of each month:

February 7, 1996,
 March 7, 1996,
 April 7, 1996,

and so on. Clearly if each payment is due one month after the previous payment, the day of the month will always be the same; the year will change only once every 12 lines (so don't worry about that for now). Only the month changes on each line, and it is dependent on the previous line. One algorithm for writing the series is:

Write the first due Date.

For each following due Date:

copy the day from the previous due Date,

write the month as the successor (the one following) of the previous month,

copy the year from the previous due Date.

This has the structure of an iterative definition. The built-in functions for the individual can be recursively filled down the column:

Feb 7, 1993	Feb 7, 1993
=DATE(YEAR(E28),MONTH(E28)+1,DAY(E28))	Mar 7, 1993
=DATE(YEAR(E29),MONTH(E29)+1,DAY(E29))	Apr 7, 1993
=DATE(YEAR(E30),MONTH(E30)+1,DAY(E30))	May 7, 1993
=DATE(YEAR(E31),MONTH(E31)+1,DAY(E31))	Jun 7, 1993

Exercises

16.1 Repeat your own name down the left side of a spreadsheet.

16.2 Write iterative algorithms for calculating the following series of numbers:

- (a) 1, 2, 4, 8, 16, 32, . . .
- (b) 2, 4, 6, 8, 10, 12, . . .
- (c) 1, 3, 9, 27, 81, . . .
- (d) 1, 0.5, 0.25, 0.125, . . .

(Hint: first figure out the relationship between each pair of consecutive items in the list; then write the algorithm.)

16.3 Build spreadsheets corresponding to each algorithm in Exercise 16.2.

(table continued on next page)

(table continued from previous page)

16.4 Write iterative algorithms for generating each of the following nonnumeric lists:

- (a) The days of the week.
- (b) The letters of the alphabet.
- (c) The sequence a, aa, aaa, aaaa, . . .
- (d) The sequence zzzzzzzz, zzzzzzzz, . . ., z.

16.5 Build spreadsheets corresponding to each algorithm in Exercise 16.4.

16.6 Build an iterative function that lists all of the days in a month.

16.7 The natural numbers are so called because they can be thought of as naturally occurring, or because they can be generated through a very simple set of axioms, called Peano's Axioms.¹ Two of the axioms are:

1 is a natural number.

*If x is a natural number
then $x + 1$ is also a natural number.*

Use Peano's axioms to generate a list of the first 100 natural numbers.

16.2

Recursively Defined Functions

Section 16.1 introduces a subtle new idea. In most previous examples, each function to be iterated down a column was defined in terms of other cells in the same row. Similarly, functions iterated across a row were defined in terms of other cells in the same column. In each vertical column, each value was calculated from a

unique set of values in the same row, but not in the same column (usually to the left). (Take a moment and check that this is true.) The independent values were usually raw data (entered by the user), and certainly not involved in the iteration itself. The calculation of each cell was based on unique initial data. The total purchases for each roommate were calculated from the individual purchases of that roommate. The iterative definition simply said that each cell was defined in the same manner but apparently always in terms of its own unique set of independent data.

The arithmetic series examples contained a subtle difference: each element is defined iteratively in terms of earlier instances of the same function: each row number is defined in terms of a previous row number. In fact, the calculations of a given row number depend on all of the previous row numbers. If any single row were removed, the definition would break down:

1. Named after the Italian mathematician Giuseppe Peano (1858-1932), who first enunciated them.

	item number
1	
2	1
3	=A2+1
4	
5	=A4+1
6	=A5+1

item number
1
2
1
2

Functions which depend on (all of) the previous rows can often be especially useful for calculating a single final value, which appears in the final row. For example, the product $a \times b$ could be defined as repeated additions:

Start with an “answer” of 0.

Repeat a times

add b to the current answer.

which can be demonstrated in a spreadsheet with the concrete example: 5×3 :

1	0
2	=A1+3
3	=A2+3
4	=A3+3
5	=A4+3
6	=A5+3

0
3
6
9
12
15

Defining a function in terms of previous instances of the same function is called *recursion*, a powerful computing tool. For example, the function $\text{factorial}(n)$ is defined as “the product of all the integers from one up to n ,” usually written as:

$$\text{factorial}(n) = \prod_1^n i$$

which is just a formal way of writing2:

$$1 \times 2 \times 3 \times \cdots \times n$$

The straightforward algorithm for creating a factorial table looks something like:

For each line in the table:

number the line.

find the product of all the numbers from 1 to the line number.

and can be coded in a spreadsheet as in Figure 16.1 on page 276. This definition does not lend itself to iteration. The defined cells do not have identical forms. Each

2. The Greek letter P (pi) is used for products in the same way that S is used for summations.

	A	B
1	number	factorial
2		=1
3	=A2+1	=1*2
4	=A3+1	=1*2*3
5	=A4+1	=1*2*3*4
6	=A5+1	=1*2*3*4*5

Figure 16.1
Calculating a Factorial

definition requires one more operation than the previous cell. No use of Fill down can create the needed cell definitions. Therefore, the programmer would need to fill in each row explicitly. However, a new perspective can reduce this problem. Notice that each definition after the first actually contains the definition above it:

$$\begin{aligned} \text{factorial}(2) &= 1 \times 2 = \text{factorial}(1) \times 2 \\ \text{factorial}(3) &= 1 \times 2 \times 3 = \text{factorial}(2) \times 3 \\ \text{factorial}(3) &= 1 \times 2 \times 3 \times 4 = \text{factorial}(3) \times 4 \end{aligned}$$

That suggests an alternative algorithm:

For the first line:

$$\text{factorial}(1) = 1.$$

For all other lines with line number, n, greater than 1

$$\text{factorial}(n) = \text{factorial}(n - 1) \times n.$$

which can be written in a spreadsheet as:

	A	B
1	number	factorial
2		=1
3	=A2+1	=A3*B2
4	=A3+1	=A4*B3
5	=A4+1	=A5*B4
6	=A5+1	=A6*B5

	A	B
1	number	factorial
2		1
3	2	2
4	3	6
5	4	24
6	5	120

The identical definitions for all cells after the first can clearly be iterated down. Such recursive definitions occur throughout computer science, with a general structure:

Base:

*For the first item
record a base or starting value.*

Step or increment:

*For all other items
perform a simple calculation based on the previous line.*

This can be stated more generally as:

*If a cell has a trivial representation
then write down the trivial representation
otherwise find a representation in terms of previous cells.*

This definition implies:

*The solution of a complicated problem is easy if you know the
solution to a slightly simpler problem.*

You can calculate the factorial of any number if you know the factorial of the previous number. The value for each row in the multiplication or factorial calculation assumed the successful calculation of the previous row.

The humble technique of assuming partial solutions is a surprisingly powerful problem-solving tool. Consider the following fable:

A peasant saved the king's life. In appreciation, the king told him to name his own reward. He didn't ask for a fortune. Instead he asked for a series of apparently small payments: one penny³ on the first day, to be followed by two pennies on the second day, four on the third, and so on. On each succeeding day, the daily payment doubles compared to the previous day. The humble man only wanted to receive this reward for one short month.

The moral of the original fable is found in an analysis of the monetary reward; the computer science moral is found in the methodology. Since the reward on each day depends on the reward

of the previous day, you can calculate its value using the following recursive algorithm:

Calculate the reward for day #1 (actually just write it down as 1¢).

For each of the following days (days #2 through #30):

calculate that day's reward as: $2 \times$ reward for previous day.

This algorithm has the appropriate format for a recursive function: a base (1¢) and a simple step (double the previous day). The corresponding spreadsheet looks like:

3. In the original fable, he asked for a grain of rice. The updated version creates a context with which the average American can more easily identify (how many grains of rice in a pound?). The principle is identical.

	A	B
1	Day	reward
2	1	\$0.01
3	=1+A2	=2*B2
4	=1+A3	=2*B3
5	=1+A4	=2*B4
6	=1+A5	=2*B5
7	=1+A6	=2*B6

	A	B
1	Day	reward
2	1	\$0.01
3	2	\$0.02
4	3	\$0.04
5	4	\$0.08
6	5	\$0.16
7	6	\$0.32

Each of the rows after the first is defined in terms of the previous row. In column B, each cell is defined as twice the cell directly above it. (Column A should look familiar.) A recursive definition is usually the easiest way to create any such series of values. Checking the results for the first few days shows that the algorithm seems to be working. And by the way, it looks like our peasant did OK for himself, as can be seen from the last few rows of the table:

26	25	\$167,772.16
27	26	\$335,544.32
28	27	\$671,088.64
29	28	\$1,342,177.28
30	29	\$2,684,354.56
31	30	\$5,368,709.12

The above fable is, of course, apocryphal. So a fair question might be:

Are there real uses for recursively defined tables?

Consider a variation on the peasant problem. Suppose that you had \$1000 to spend now. You might want to invest it for your retirement. That sounds like a crazy idea after all, retirement is probably 45 years away. But let's see what happens. Clearly, no legal investment doubles every day like the peasant's windfall. So suppose you invested it in the stock market. Historically the return on stock market investments has averaged about 11% per year. That is, each dollar invested at the beginning of a year grows to \$1.11 at the end of that year (on the average). If you do not spend any of the profits, the entire \$1.11 is available to invest the following year (this is called *compounding*). The method for calculating the total is

almost identical to that of the peasant's fortune:

4. A series of this sort is called a *geometric* series.

Calculate the amount invested in year # 1

(actually just write it down as \$1000).

For each of the following years (years #2 through #46):⁵
calculate the total available after that year as:

this year = last year \times 1.11 (or last year + (0.11 \times last year))⁶

The investment seems to start out slow enough taking a full 7 years to double.

	A	B
1	year	total return
2	1	\$1,000.00
3	=1+A2	=1.11*B2
4	=1+A3	=1.11*B3
5	=1+A4	=1.11*B4
6	=1+A5	=1.11*B5
7	=1+A6	=1.11*B6
8	=1+A7	=1.11*B7
9	=1+A8	=1.11*B8

	A	B
1	year	total return
2	1	\$1,000.00
3	2	\$1,110.00
4	3	\$1,232.10
5	4	\$1,367.63
6	5	\$1,518.07
7	6	\$1,685.06
8	7	\$1,870.41
9	8	\$2,076.16

But by retirement time, 45 years later, it appears you will be doing just fine (not quite as well as the peasant, but pretty well for the size of your investment):

42	41	\$65,000.87
43	42	\$72,150.96
44	43	\$80,087.57
45	44	\$88,897.20
46	45	\$98,675.89
47	46	\$109,530.24

5. Notice that starting with year #1 and going for 45 more years gets to year #46. This is the sort of situation that causes computer scientists to prefer to call the first number in a series 0, followed by #1, helping avoid fence post errors (see Box 15.2 for other examples).

6. There is actually a shorter method for calculating the total return. The amount after 45 years can be described by:

$$\text{final investment} = \text{initial amount} \times e^{45 \times \log(1 + \text{rate})}$$

But, then, I did promise you: no complicated mathematics.

Exercises

- 16.8 Build the peasant spreadsheet, but add one more column. The amounts in the spreadsheet represent daily rewards. But what is the total reward? First, write an English description of the total reward based on the amount received as of the day before. Then, enter it into the spreadsheet.
- 16.9 Compare the result of Exercise 16.8 to a realistic salary (say \$2000 per month). How many days will it take until the peasant's total reward equals his salary for the same number of days?
- 16.10 Write algorithms for calculating the following series of numbers:
- $$1, 1, 2, 3, 5, 8, 13, 21, \dots$$
- This series is called the Fibonacci7 series. Each number (after the second) is the sum of the two previous numbers.
- 16.11 Major credit cards allow you to pay off your loan slowly. A typical card allows you to pay back only 6% per month (with usually a minimum payment of something like \$20). And they allow you to do it while you continue to spend. Of course they charge for this service: typically 1.5% per month (wow, that's 18% per year). If you spent \$1000 using your charge card, and started paying it off at their minimum rate, how long would it take you to pay off the full \$1000? How much would you pay the bank in interest during that time? How long will it take if you also spend \$10 in new purchases each month?
- 16.12 Most people use the prefix *kilo* to mean 1000 (see Box 3.2). When computer scientists use the term, they usually mean 1024. Use the series in Exercise 16.2(a) to hypothesize a reason for this discrepancy. An analogous discrepancy exists

for the prefix *mega*. What do you suppose it is?

16.13 Contrast the geometric series with the arithmetic series introduced in the preface.

16.14 Section 12.2.1 suggested that you, the programmer, should always suspect that your program is wrong. In light of the surprising results based on the "humble peasant" fable, how should you convince yourself that the spreadsheet results are, in fact, accurate?

16.2.1

A Hybrid Example

Some functions are recursively defined, but also refer to entries in the same record or line. Consider the task of keeping the register for a checking account (the record of the checks actually written). The balance after each check can be described as:

7. After the Italian mathematician and early population biologist Leonardo Fibonacci (d. ca. 1250).

new balance = old balance - amount of this check

The new balance is defined in terms of the previous entry but it is also defined in terms of an item in the current record. Nonetheless, a complete recursive description of the final balance is easy:

The first line contains the initial balance.

For all later entries:

new balance = old balance - amount of the current check.

The corresponding spreadsheet is straightforward:

	A	B	C	D
1				
2	check number: to		amount	balance
3	initial balance			\$1234.00
4	1 ABC auto	\$135.17	=D3-C4	
5	=1+A4	CDE wrecking	\$42.00	=D4-C5
6	=1+A5	XYZ auto	\$0.17	=D5-C6
7	=1+A6			

C	D
amount	balance
	\$1234.00
\$135.17	\$1098.83
\$42.00	\$1056.83
\$0.17	\$1056.66

Other Recursive Functions.

Function definitions such as the above, based on the previous value and a newly entered value, are also very common. Consider a few more examples:

On a pay stub:

*year-to-date earnings =
previous year-to-date earnings + amount of this paycheck*

A credit card monthly statement is based on

this month's total = last month's statement + this month's charges

Any “running total” is based on

the new running total = the previous running total + the new item

Most bookkeeping techniques.

A multiple semester transcript, showing the cumulative GPA after each semester.

Exercises

16.15 Build a check ledger, but include deposits as well as withdrawals.

16.16 Build a spreadsheet corresponding to each of the bulleted items above.

16.3

Relative Versus Absolute Addressing

There are two ways to point to or describe locations: by some *absolute* or fixed system, or *relative* to some other reference point. Humans commonly use both methods. Generally if you are asked for directions to some location, you give directions relative to some specific location: “Go three blocks (from here) and turn right.” Unfortunately, this will get you to the desired location from one starting point but not from another. Similarly, “The desk is to your right,” describes the location of the desk relative to you (your current location). On the other hand, although “Where is New York City?” can be answered with “It’s at $40^{\circ} 43' \text{ North}, 74^{\circ} 1' \text{ West}$ ” (a location in an absolute set of coordinates), most people would not find this a very useful description. At first, it may feel as if the relative directions are always best. However, the best choice will depend on the current use. “Where is Professor Dingbat?” “He is in Ivory Hall, Room 123.” This answer gives his location in terms of a fixed location (an absolute address). The corresponding relative address might be a set of directions for getting to his office. The absolute location is useful to anyone with a good knowledge of the campus, but relative directions might be better for a visitor.

Many computer applications allow both relative and absolute addressing. A word processor may allow the user to scroll ahead or back one screen or one page at a time. That is, in effect, “go to the next page” a relative address. The page pointed to by the phrase “next page” depends on the page currently displayed. Some applications also accept the command, “Go to page 17.” Clearly that is an absolute address. Page 17 is the same page, no matter what page is currently visible.

The use of iteration in a spreadsheet requires the user to distinguish between relative and absolute addresses. The first line of the table of squares example contains:

this cell = cell B2 × cell B2

Each subsequent line computes:

this cell = the cell to the left × the cell to the left

not

this cell = cell B2 × cell B2

In theory the first line could be defined with either an absolute or a relative address. An extension of the definition to remaining cells should say, “define this cell in the same way as the one above.”

Defining the first cell absolutely in terms of B2 creates a problem: all the remaining cells will get exactly the same definition and meaning: B2 * B2. Each cell should actually be defined in terms of the cells to its own left, not to the left of the first cell i.e., a relative address. The designer of the algorithm (i.e., you) must specify which type of addressing is needed.

In contrast, consider the last steps of the roommate expense calculation.

*For each roommate,
the amount owed is the difference between
the amount that person paid and the average.*

Notice that it is the same average for each person. Using this definition iteratively requires an absolute address: each line must refer to the same average. The multiplication table also needs absolute addresses: referring to explicit rows and columns.

Relative Versus Absolute Addressing in a Spreadsheet.

Fortunately, most spreadsheets provide a simple straightforward solution to this problem: two notations, one for relative and one for absolute addresses. On the surface, the standard notation used for defining functions in a spreadsheet appears to be absolute: a function refers to other cells explicitly by row and column. In contrast with the wording used in most iterative algorithms in this chapter, a spreadsheet has no words like “above,” “previous,” or “left,” or “over two positions.” Spreadsheets distinguish between relative and absolute addresses in a seemingly odd manner. Iteration is so common, that the `Fill down` command regards the default notation as relative addressing, even though it appears to be absolute. Any definition that depends on

“the items to the left of this cell” or

“the item above this cell,”

is treated as a relative address. Thus, most previous examples require no changes. In contrast, the designer must explicitly indicate each absolute address. Typically, absolute addresses are marked with a special character, such as a “\$” preceding the reference: `$B3`. Most spreadsheets allow the designer to specify absolute or relative addressing for rows and columns independently; for example, it might use:

`$B3` means “when iterating, keep the column constant” (i.e.,

column B is referenced absolutely).

B\$3 means “keep the row constant” (i.e., row 3 is absolute).

\$B\$3 must mean that “both dimensions are absolute references.”

Example 16.2 You want to know how much sales tax would be due on each of several possible purchases. Clearly you could make a table showing the price and tax on each item, as in Figure 15.4. The tax is computed in the same way for each, so you obviously want to define the table iteratively:

For each item:

tax on that item = tax rate × cost of the item.

8. The specific notation may vary from one product to another. For example one vendor may use a “#” rather than a “\$”. The principle remains the same: mark the absolute references.

Notice that the tax rate is constant for the entire problem (it may go up next week, but it will be the same on all items you are comparing today). It seems wasteful to repeat the tax rate on each line as in the earlier example. Instead, each calculation should refer to a single tax rate. This requires an absolute address, as in Figure 16.2. A relative address is still needed to refer to the cost of individual items.

The general rule of thumb for addressing is:

*If all lines should refer to exactly the same cell,
then use absolute addressing.*

*If each line should refer to unique other cells
then use relative addressing.*

Exercises

16.17 Build the multiplication table described in Section 15.3.4 using a combination of relative and absolute addresses.

16.18 Repeat Exercise 16.17 using only relative addresses, but a recursive definition. Hint: Peano's definition of multiplication includes:

```
if b = 1
  then a × b = a
if b > 1
  then a × b = a × (b - 1) + a.
```

16.4

Recursion and “What-if” Problems

Many problems can be stated in the form:

*Some event occurs every so often.
How long will it be until . . .*

For example, the sports playoff tree of Section 6.2.3 graphically displays the scheduled matches in a tournament. The designers of

such a tournament must start with one basic question: “For the number of teams that we expect, how many rounds must the tournament have?” This can also be stated as, “How high is the

1	sales tax calculations	tax rate =	7.00%
2	item	cost	tax
3	fur coat	\$1000.00	=B3*D\$1
4	apple pie	\$3.15	=B4*D\$1
5	new shoes	\$49.95	=B5*D\$1

tax rate =	7.00%
tax	
\$70.00	
\$0.22	
\$3.50	

Figure 16.2
Use of an Absolute Address

tree?"⁹ Fortunately, a spreadsheet provides a simple solution: since each round eliminates half the teams, just divide the number of teams left by 2 until there is only one left. Figure 16.3 shows the calculation for the number of rounds in the 64-team NCAA Basketball championships. The programmer did not need to know how many lines to create. Any extra lines yield nonsense answers such 0.5 teams. Also notice that, effectively, "round 0" is the point at which there are 64 teams. Similar "how long" problems include

How long will it take to pay off my credit card?

How many things on my wish list can I buy?

How many of my kids can fit in the boat (without being too heavy)?

Exercises

16.19 Suppose you invest \$4000 at 8% annual interest. How long will it take to double your money? (Hint: see Section 16.2.)

16.20 Extend Exercise 16.19 to a "what-if" table asking, "what if you could get 5, 6, 7, 8, 9, 10, or 11% interest?"

16.21 Build a recursive function that lists all of the dates in a month.

A	B	C
The NCAA basketball tournament		
round	teams remaining	
number	after this round	
start	64	
1	32	
2	16	
3	8	
4	4	
5	2	
6	1	
7	0.5	
8	0.25	
9		

A	B	C
The NCAA basketball tournament		
round	teams remaining	
number	after this round	
start	64	
1	=B4/2	
=1+A5	=B5/2	
=1+A6	=B6/2	
=1+A7	=B7/2	
=1+A8	=B8/2	
=1+A9	=B9/2	
=1+A10	=B10/2	
=1+A11	=B11/2	
=1+A12		

Figure 16.3
Basketball Tournament

9. The mathematical solution for this question is given by

$$\text{number of rounds} = \lceil \log_2(\text{number of teams}) \rceil$$

But once again, I promised, “No tricky mathematics.” Many related examples can be solved either with a similar mathematical formula or with a recursive algorithm.

16.5

Iteration as a Model of Analytic Integration

The integral of a number is often described as the area under the curve described by the function. Most introductory calculus courses ask students to approximate this area using *Riemann sums*,¹⁰ defined something like:

$$\sum \Delta x \times f(x)$$

That is, it is the sum of the areas of many thin rectangles, each Δx wide and $f(x)$ high. The value $f(x)$ of course is different for each x . This is a problem ideally suited for an iterative spreadsheet. Figure 16.4 shows the approximation of:

$$\int_0^2 x^2 dx$$

	A	B	C	D
1	calculating an integral of	x^2		
2	delta x		0.01	
3	from		0	
4	to		2	
5	x	f(x)	delta x * f(x)	running sum
6	0.01	0.0001	0.000001	0.000001
7	0.02	0.0004	0.000004	0.000005
8	0.03	0.0009	0.000009	0.000014
9	0.04	0.0016	0.000016	0.00003
...				
199	1.94	3.7636	0.037636	2.452645
200	1.95	3.8025	0.038025	2.49067
201	1.96	3.8416	0.038416	2.529086
202	1.97	3.8809	0.038809	2.567895
203	1.98	3.9204	0.039204	2.607099
204	1.99	3.9601	0.039601	2.6467
205	2	4	0.04	2.6867

Figure 16.4
Calculating a Riemann Sum

- After the German mathematician, Georg Friedrich Bernhard Riemann (1826-1866), who invented the technique.

using a Dx of 0.01. This task would be time consuming, to say the least, even with a calculator. The Riemann sum comes within about 0.01 of the integral found by analytic method. In fact, scientists and mathematicians use techniques very similar to this whenever they need to calculate integrals using a computer.

Exercises

16.22 Find the integral of x^3 .

16.23 In Exercise 16.22, how small a D is needed to get an answer within 0.001 of the correct answer?

16.6

Summary

Iterative algorithms are among the most powerful tools available for any computer user. Iterative functions in which each instance is defined in terms of previous or simpler instances are called *recursive functions*. Implementing iterative and recursive algorithms is made simpler through the use of absolute addresses. Iterative approximation forms the basis of many analytic mathematical techniques used in the sciences.

Important Ideas

recursion	arithmetic series	geometric series
factorial	absolute address	relative address

Riemann sum

17

Conditional Actions

If my husband would ever meet a woman on the street who looked like the women in his paintings, he would fall over in a dead faint.

MRS. PABLO PICASSO

Pointers

Gateway Conditionals

Lab:

Lab Unit 10:

Manual: Conditionals

17.0

Overview

The solution to many problems depends on circumstances unknown at the time the algorithm is written. For example, the answer to the question, “What is the best route to the shopping mall?” depends on the questioner’s current location (or location when the trip actually begins). There is no way for an advertiser to know where the customer will be or more importantly, where every customer will be at the time she reads the directions. But the definition of algorithm requires a solution which will always work. In everyday transactions, people get around this problem by any of several methods:

1. Provide multiple sets of directions, one for each place from which the shopper may wish to begin the trip.
2. Provide directions from some general well-known starting point.
3. Begin by asking, “Where will you be coming from?” and then

provide the appropriate set of directions.

None of these approaches seems to match the definition of algorithms exactly. *Logic* and *conditional actions* are algorithmic tools for dealing with multiple alternatives.

17.1

Algorithms with Alternatives

The third approach in the preceding list is closest to the algorithmic method. The first two approaches both require the shopper to do part of the problem solving

selecting the appropriate algorithm or getting to the well-known starting point. The first approach also seems to require finding several distinct solutions to the problem instructions, which seems especially wasteful for two locations that are very close to each other. Consider an example that seems slightly more amenable to algorithmic solution.

Example 17.1 The Itty Bitty Machine Co. sells computer equipment. Itty Bitty's management believes the customer is the most important person. Therefore, they discount all purchases. But since they also want to make big sales, the amount of the discount depends on the size of the sale:

sales over \$100.00 (before the discount) receive a 20% discount.

sales up to \$100.00 get only a 10% discount.

One algorithm for calculating the final price is:

Find the total (prediscount) price of the purchase.

Find the appropriate discount rate.

Calculate the discount amount.

Subtract the discount from the purchase price.

The second step may seem to defy algorithmic treatment. Sometimes the cashier should use one value, sometimes another. The designer cannot specify the correct rate since the size of a given sale is not known at the time the algorithm is written. It seems inappropriate to ask the customer which discount to use, and it seems silly to have a large chart showing the after-discount price for every possible purchase. The designer needs a way to express:

If the purchase price is greater than \$100.00

then the discount rate is 20%

otherwise the discount rate is 10%

and substitute this expanded step for the middle step in the original algorithm. Such a step is called a *conditional* because the action taken depends on other values or conditions; execution of either alternative is *conditional* on the question posed. Fortunately, every programming application provides conditional tools.

17.1.1

Anatomy of a Conditional

A conditional step in any algorithm has three basic parts:

A *test* or question that can be answered yes or no.

An action to perform if the answer to the test is yes.

Another action to perform if the answer is no.

In all other respects a conditional is just like any other step in an algorithm. Consider each of the parts.

The test: A useful test has three characteristics. It is a “yes/no” question. Generally computer scientists use the slightly more formal words *true* and *false*, which make it easier to talk about the question and the answer. The test must be answerable with information available at the time the step is executed, but generally this answer cannot be known at the time the algorithm is written. For example, “Is the price greater than 100.00?” is answerable once the price is known, but not until then. The programmer must recognize the need for the test but need not know what the price will be. For numeric data, the test might ask:

Are the two values equal?

Are the two values unequal? (which is almost the same question)

Is the first value greater than the second?

Is the first less than the second?

The two actions: Any legal (well defined) algorithmic step can serve as an action. “Well defined” means exactly what it has previously: unambiguous. Two common actions are:

display a simple value, and

compute a new value.

In the example, the possible discount rates are each simple numbers:

20%

and

10%.

17.1.2 *Representation of Questions*

Mathematicians and computer scientists use a shorthand representation for algorithmic questions. Each question contains two values to be compared and an operator specifying the comparison. The question

Is the price greater than \$100.00?

from Example 17.1 becomes:

price > 100.00

It is just a short form of the question, containing no supporting English phrases, modifiers, qualifiers, or question mark. The common tests are the standard arithmetic relations, represented in the common arithmetic notation:

Are they equal?	item1 = item2
Is the first greater than the second?	item1 > item2
Is the first less than the second?	item1 < item2
Is the first either greater than or equal to the second?	item1 ³ > = item2
Is the first either less than or equal to the second?	item1 [£] < = item2
Are they unequal?	item1 ¹ < > item2

The right-hand representations are variations on the traditional mathematical symbols ³, £, and ¹, used because the traditional symbols are not found on most keyboards. Notice that each of those is actually a combination of two tests. Even the odd symbol “<>” actually expresses the concept of not equal: if two numeric values are not equal, the first must be either smaller (<) or larger (>) than the second. The symbol for not equal, “<>”, is a combination of the two, meaning “less than” or “greater than.” The existence of compound symbols suggests that such combination questions are very common. Chapter 18 discusses other forms of combined questions.

Although the mathematical notation is more concise, its principal advantage is that it forces the designer to be more specific or less ambiguous. There are no comparisons such as “approximately equal.” The notation also does not lend itself to non-words such as “the appropriate discount.”

17.1.3 *Conditionals in a Spreadsheet*

If it keeps up, man will atrophy all his limbs but the push-button finger.
FRANK LLOYD WRIGHT

The representation of a conditional in a spreadsheet cell has each of the three basic parts, plus some “syntactic glue” to distinguish the parts and connect them all together.

The Test.

All spreadsheets use the mathematical notation for comparisons. The values can be either constants or cells containing values. If `price` in the above example is contained in cell C2, then the question

```
price > 100.00
```

becomes:

```
C2 > 100.00
```

The Actions.

Actions may be either “display a value” or “calculate a value.” In the example, both actions are of the form “display a value,” either 10% or 20%.

Syntactic Glue.

All that remains is a bit of syntax to hold the three phrases together. The exact details of this syntax vary among the different spreadsheet applications, but the basic structure will always be similar to that shown in Figure 17.1. The glue serves two distinct purposes:

to indicate that the cell has a conditional definition (by the word `if` in the figure)

to segregate the various components: the test and the two actions.

Notice from the form of the conditional, that it appears to be a built-in function called `if`. It takes three arguments: the test, the “yes” action, and the “no” action, in that order. Separate the three parts with commas and group them together by enclosing them in parentheses, as you would any built-in function. Figure 17.2 on page 294 shows the discounts defined conditionally.¹

Choosing a Cell’s Content.

The values displayed can be values held in other cells. For example, the maximum of two values can be found using the algorithm:

*if the first > the second
then display the first
otherwise display the second.*

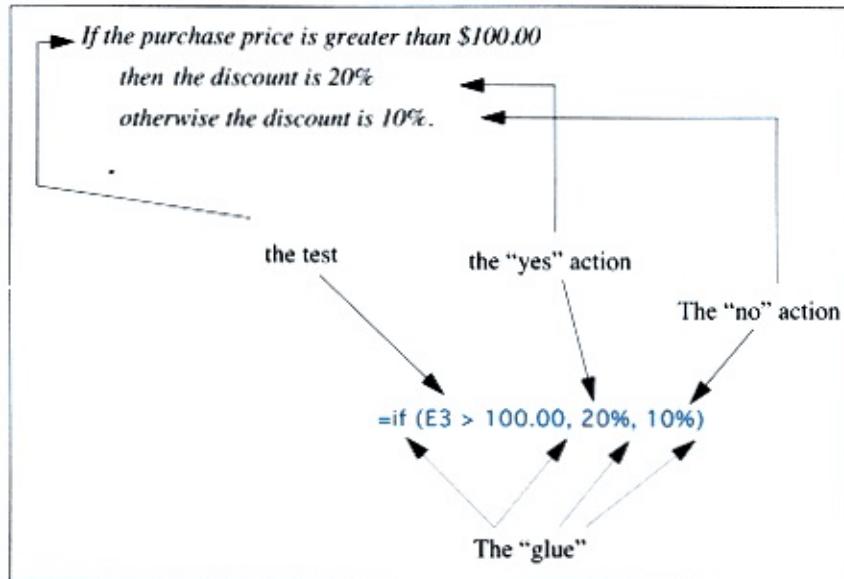


Figure 17.1
Spreadsheet Representation of a Conditional

1. Although the syntax described here for conditionals is almost universal, some spreadsheets may differ in the format of the individual values. For example, some may require that you write just the percentage without the “%”. Use your application’s online help to see the exact form it uses.

D2		<input checked="" type="checkbox"/>	<input type="checkbox"/>	=IF(C2>100,20%,10%)	
		A	B	C	D
1					
2				\$110.00	20%
3				\$90.00	10%

Figure 17.2
A Conditional Discount Based on Price

In a spreadsheet, with values in cells A3 and A4, this may look like:

= IF (A3 > A4, A3, A4)

Notice that there is no way to say "the one I just mentioned." To reference "the first" (A3), the designer must explicitly name the cell again. This helps reduce ambiguity.

Conditional Calculations.

The value placed in the cell by a conditional can also be a calculated value. The algorithm may have two competing calculations:

Calculate a value by method 1.

or

Calculate a value by method 2.

For example, a vendor may sell a product for \$2 each or \$1.80 each for purchases of more than five of the product. An algorithm for this is

*If number purchased £ 5
then price = 2 × number purchased
otherwise price = 1.80 × number purchased.*

which in a spreadsheet with the number purchased in cell B5 becomes:

= IF (B5 < = 5, 2 * B5, 1.80 * B5)

Notice that although the general definition of *conditional* seems to allow any action at all, in a spreadsheet the action is restricted to computing and displaying a value. In addition both actions must refer to the same cell (the one holding the condition). Also notice how the use of the mathematical expression helped clarify the original English. Some people get confused about what would happen if exactly 5 items were purchased. The written expression dictated the interpretation: the first action was performed if the number was less than or equal to 5. The second action was performed only if the number purchased was actually greater than 5.

Conditional Definitions in Multiple Cells.

More than one cell may depend on a given comparison. For example, a better version of the algorithm for the housemate problem might say:

Calculate amount this person owes to others.

or

Calculate amount others owe to this person.

The algorithm designer must determine the conditions under which each of the two calculations should be performed. For example:

If this person owes money

then calculate amount this person owes to others

otherwise calculate amount others owe to this person.

Since the result may represent either of two distinct calculations, the result should have a label indicating which it is. Two actions will be required:

If this person owes money . . .

then write "owes"

calculate: amount owed by others

otherwise write "is owed"

calculate: amount owed to others.

Notice that in either case, the amount should be a positive number. A person owes money if he or she has paid less than the average paid by all individuals. The test can be:

If this person has paid less than the average paid by everyone . . .

Combining these actions gives:

If this person paid less than the average paid by everyone

then write "owes"

calculate: average - amount paid

*otherwise put “is owed”
calculate: amount paid - average.*

The corresponding spreadsheet requires two conditionals, one in the cell that will contain “owes” or “is owed,” and one in the cell that will contain the dollar amount, as in Figure 17.3 on page 296.

17.2

Arithmetic Relations as Functions

Some readers may have noticed an apparent inconsistent usage of the term “relation”. Section 8.1 defined “relation” as a set of ordered pairs. In Section 17.1.1, arithmetic relation seems to refer to the answer to a “yes/no” question. Technically, the original definition is correct. The latter is really a shorthand

	A	C	D	E	F	G	H
1	Expenses for the month of October						
2	who	purchases				Totals	
3	Chris	\$5.00	\$20.00	\$13.00	\$92.00		\$130.00
4	Pat	\$437.00					\$437.00
5	Joe	\$37.00	\$37.00	\$37.00	\$37.00	\$37.00	\$185.00
6							
7	average						\$250.67
8	Summary						
9	Chris	is owed	\$120.67				
10	Pat	owes	\$186.33				
11	Joe	is owed	\$65.67				

8	Summary						
9	Chris	=IF(H2>H\$7, "is owed", "owes")	=IF(H3>H\$7, H3-H\$7, H\$7-H3)				
10	Pat	=IF(H3>H\$7, "is owed", "owes")	=IF(H4>H\$7, H4-H\$7, H\$7-H4)				
11	Joe	=IF(H4>H\$7, "is owed", "owes")	=IF(H5>H\$7, H5-H\$7, H\$7-H5)				

Figure 17.3
Roommate Example with Conditional Labels

consistent with common English usage. The relation “less than” is a set containing all pairs of numbers, such that the first is less than the second:

Less than

1	2
2	3
1	3
1	4
2	4
...	...

Since there are an infinite number of such pairs, you can't actually list the elements of the “less than” relation. The usage in this chapter really asks:

*if a given pair of numbers is in the relation “less than”
then answer “true”
otherwise answer “false”*

That is, if we actually had such a list, would the pair be on the list?
Thus

$$2 < 3$$

is true, because the pair (2,3) is in the relation “less than”.

The relation “less than” can also be thought of as a function from a pair of numbers to the values `true` or `false`:

less than: (numbers × numbers) ® {true, false}

Like any built-in function, they can be used to define cells within a spreadsheet, for example

= (2 < 3)

or even the equivalent, but strange looking²

= 2 < 3

Both yield the answer `true`. Finally, notice the parallel structure between these binary relations and the binary arithmetic operators (+, -, ×, ÷). Each operator sits between two values rather than in front of the values as in other built-in functions such as `sum(1, 2, 3)`.

Exercises

17.1 Test each of the arithmetic operators: `<`, `>`, `=`. Check enough values to verify that they behave as you expect.

17.2 Suppose sales tax is defined as 7.5% of the purchase price, but all purchases of less than \$0.20 are exempt from tax.
Write and test a spreadsheet expression that will calculate the correct tax.

17.3 Write an expression to find the minimum of two numbers.

17.4 Write an algorithm that draws a square or a triangle based on the user’s request.

17.5 The comparisons `<=` and `>=` are actually redundant. Write alternate expressions equivalent to each of the following and test your results in a spreadsheet.

a < = b
a > = b

17.3

Exploring Conditionals

*If thou art a master,
be sometimes blind;
If a servant,
sometimes deaf.*

THOMAS FULLER

2. Recall the suggestion that you use parentheses to disambiguate arithmetic expressions. It holds here too.

The conditional is a surprisingly useful operation. At first glance some users may say: I don't want to deal with such complicated situations. But the tool is so useful, it should not be ignored.

17.3.1

Uses for Conditionals

Conditionals have uses in more situations than just complicated arithmetic problems.

Data Versus Blanks.

Some situations provide no usable answer. In those cases, it is often better to leave the field blank than to provide a "garbage" result. For example, if a customer pays too much, the line

```
payment due: -$10.00
```

would probably be better written as

```
payment due:
```

or

```
payment due: none
```

Otherwise the customer is likely to think that \$10.00 is the amount due. The conditional:

If amount owed > 0

then display the amount owed

otherwise display "nothing due"

provides this service. Notice that, in general, there is no rule requiring a given cell to always contain the same type of information, e.g., always a number, or always characters. However, if other cells refer to this value in a calculation (perhaps to calculate a minimum payment), the type should be consistent. An attempt to perform the calculation: 5 - "nothing due" will generate an error.

Error Diagnostics.

Most applications protect the user from errors. For example, if the definition of a cell causes some value to be divided by zero, the application will generate an error message: something like “zero divide” (or a condensation of that: `zerodiv`). Unfortunately, that message tells the user what the error was, but not why there was an error or what caused the error. It is up to the programmer to figure out why the divisor was zero. In a complicated spreadsheet, this may require checking the value in several other cells and thinking through the logic involved.

A much better time to deal with the problem is before the error occurs. The programmer can identify situations that might create potential problems and print a more meaningful error message. For example, suppose an algorithm included the general steps:

Count the number of items.

Find the total value of all the items.

Find the average of the values.

Clearly the average could be found by

average = sum/number of elements.

If, for some reason the number of items was zero, the quotient would be undefined. If D4 and D5 contain the sum and number of items respectively, a conditional such as

```
If (D5 > 0, D4/D5, "No elements!")
```

would calculate the quotient when it is defined and provide a more useful message when it was not defined.

Missing Input.

Many questionnaires have optional data or data that is only relevant for some persons. For example, a medical form may have a section for females only. A woman filling out the form should fill in the appropriate data in the section. A man should leave it blank. Any program processing this information must respond accordingly. The conditional

```
if ((C5 <>""), action1, action2)
```

asks “Is there data in cell C5?” or, more specifically, “Is the value in C5 other than the null string (the string with nothing in it)?” In this case one action would be to perform a meaningful calculation. The other action should display a blank or some other message meaning “not applicable.”

17.3.2

Invisible Conditionals

Several built-in functions and formatting features of spreadsheets actually contain implied conditionals. Their use is so common that

the designers of the application provided shortcuts.

For Negative Numbers.

Positive and negative numbers often need to be treated differently, as seen in previous examples. The solution for the housemate example explicitly tested for negative numbers. Common accounting and/or programming practices provide alternatives for making the distinction either simpler or more obvious. The following all mean - \$10.00:

parentheses: (\$10.00)

red:3 \$10.004

a debit sign: \$10.00D5 (D for “Debit”)

a combination of the above: (\$10.00)

Depending on the application, one of these may be more appropriate than using conditionals.

Built-in Functions.

Many built-in functions actually make the same calculations as some of the common conditionals. For example, the maximum of two values can be found using the algorithm

```
if the first > the second
then      display the first
otherwise  display the second
```

The built-in function, `max`, accomplishes exactly the same thing. In addition, it works for more than two values. Most important, as with other built-in functions, it has been thoroughly tested, so it may save the user an error or two. Similar built-in functions with implicit conditionals include minimum, absolute value, sign, choose, or match.

Exercises

17.6 Write a function that determines if there is an input value in a cell, and prints, “Please enter a value,” or “Thank you,” as appropriate.

17.7 Repeat the “how long” problems of Exercise 16.11 and 16.19, but make sure that all unneeded cells at the end are left blank.

17.8 Build a spreadsheet that calculates a square root. Use the builtin function, but test the value of the operand first to make

sure that it is computable.

17.9 Define a cell equivalent to the absolute value builtin function.
Test your answer by comparing it to the builtin function.

3. This obviously is the source of the expression “in the red.”
4. Oh well, it isn’t red here. You get the point though: an alternate color makes the distinction stand out.
5. This convention is supported by the *COBOL* programming language. Obviously single-color printers cannot print negative values in red. The single character “D” takes less space than the pair of parentheses, and makes aligning columns of positive and negative numbers easier. The practice is confined to computerized accounting.

17.3.3

Conditionals with Nonarithmetic Operands

Comparison is not restricted to arithmetic operands. Any two values can be compared for equality. It is possible to ask questions of the form

Does “Bill” = “Mary”?

Is February 29, 1994 = March 1, 1994?

Does true = false?

each of which happens to be false. Note that the first question actually asks if the string of characters “Bill” is exactly the same as the string of characters “Mary”. It does not ask if the two words are just different names for the same person. The question

“Bill” <> “Mary”

asks the opposite question: “Is the string `Bill’ different from the string `Mary’ ?” Even the question

“Bill” > “Mary”

is legal, meaning “Does the string `Bill‘ come after the string `Mary‘ in alphabetical order,” (returning a result of `false`).

Notice that treating “<” as meaning “coming before when listed in the usual order,” works for either numbers or character strings: one string is less than another if it come first when written in alphabetical order; one number is less than another if it comes first when written in increasing numeric order.

Most spreadsheet applications provide similar tools for comparing dates and times. In addition, they provide other functions that return true/false values such as functions to determine if a cell contains a number (perhaps `ISNUMBER`) or if a cell contains text (perhaps `ISTEXT`). Any function that returns a true/false value can be used in a conditional in exactly the same way as the comparison operators are used.

The Arithmetic of Nonarithmetic Objects.

In addition to comparing strings for alphabetic order, it is often useful to compare their length. For example, suppose you want to know if a string is too long to fit into a cell. If you know that the cell is wide enough for 20 characters, you could create the conditional:

*if the length of the string < = 20
then print the string
otherwise print a warning message*

or in a spreadsheet:

= IF(length(C6) < = 20, C6, "too long")

Most spreadsheet applications provide similar functions for finding one string within another and for converting dates and times to their numeric (sequential) equivalent.

Exercises

17.10 For each of the following pairs, find out which comes first in your spreadsheet application.⁶ For each pair, describe the difference in English, and check your description by testing another example fitting that description.

- (a) `ax` and `axe`.
- (b) `AAA` and `aaa`.
- (c) `1` and `one`.
- (d) February 29, 1996 and March 1, 1996.
- (e) `true` and `false`
- (f) “spread sheet” and “spreadsheet”

17.4

Conditionals in More Complicated Algorithms

Any step of an algorithm may be a conditional. This means they can occur anywhere within a spreadsheet, and in any quantity. The action taken in either (or both) of the action clauses in a conditional can be any action including another conditional.

Example 17.2 Although every computer can perform basic mathematics and logic, a given application does not necessarily provide built-in functions for every combination of logic needed for each specialized problem. Suppose you are performing a scientific experiment that involves timing the duration of computer events. Your computers can record the begin and end times for the experiment, but you must provide the algorithm:

Record the start time.

Record the finish time.

Subtract start time from finish time.

It looks simple, but what happens if the start time was 4:05:17 and the finish was 4:08:00?

4:08:004:05:17

?

Subtracting 17 seconds from 0 seconds requires “borrowing.” Although this problem is familiar to all, specifying the algorithm carefully enough for a

6. The answers for some parts of this question will actually vary somewhat depending on the machine, or even the spreadsheet application that you use. Each of the examples represents a special case. Do not waste too much time memorizing the answers.

computer to follow is a little tougher. To improve the algorithm, break each of the times into three pieces: hours, minutes, and seconds.

Refinement 1.

Record the start time as $hours0$, $minutes0$, and $seconds0$.

Record the finish time as $hours1$, $minutes1$, and $seconds1$.

Subtract start time from finish time:

$$\text{elapsed seconds} = \text{seconds}1 - \text{seconds}0$$

$$\text{elapsed minutes} = \text{minutes}1 - \text{minutes}0$$

$$\text{elapsed hours} = \text{hours}1 - \text{hours}0$$

For now, look only at the computation of the seconds. If $\text{seconds}1 \geq \text{seconds}0$ there is no problem. But any problem with $\text{seconds}1 < \text{seconds}0$ requires borrowing. This is a classic conditional:

If $\text{seconds}1 \geq \text{seconds}0$

$$\text{then } \text{elapsed seconds} = \text{seconds}1 - \text{seconds}0$$

$$\text{otherwise } \text{elapsed seconds} = (60 + \text{seconds}1) - \text{seconds}0$$

The minutes also need some attention. Whenever you borrow seconds, you must also adjust the minutes by subtracting 1. Again, a conditional will do:

If $\text{seconds}1 \geq \text{seconds}0$

$$\text{then } \text{elapsed minutes} = \text{minutes}1 - \text{minutes}0$$

$$\text{otherwise } \text{elapsed minutes} = (\text{minutes}1 - 1) - \text{minutes}0$$

The final algorithm incorporating both improvements is:

Refinement 2.

Record the start time as $hours0$, $minutes0$, and $seconds0$.

Record the finish time as $hours1$, $minutes1$, and $seconds1$.

Subtract start time from finish time:

If $\text{seconds}1 \geq \text{seconds}0$

$$\text{then } \text{elapsed seconds} = \text{seconds}1 - \text{seconds}0$$

otherwise $\text{elapsed seconds} = (60 + \text{seconds}1) - \text{seconds}0$
If $\text{seconds}1^3 \text{ seconds}0$
 then $\text{elapsed minutes} = \text{minutes}1 - \text{minutes}0$
 otherwise $\text{elapsed minutes} = (\text{minutes}1 - 1) - \text{minutes}0$
 $\text{Elapsed hours} = \text{hours}1 - \text{hours}0.$

	A	B	C	D
1				
2		minutes		seconds
3	finish		5	33
4	start		3	43
5				
6	elapsed time	=IF(D3>=D4,C3-C4,C3-C4-1)	=IF(D3>=D4,D3-D4,60+D3-D4)	

	A	B	C	D
1				
2		minutes		seconds
3	finish		5	33
4	start		3	43
5				
6	elapsed time		1	50

Arithmetic Complexity in a Comparison.

The comparison may be more complicated than "A < B". The objects compared can be any well-defined expression. For example, you can compare twice one cell to the value of another:

`IF((2 * A3) < A4, ...)`

Notice the use of parentheses. Just as with arithmetic expressions, the parentheses make it clear that A4 is to be compared to $(2 * A3)$. The expression " $2 * A3 < A4$ " might be interpreted as two times the result of the comparison " $A3 < A4$ ". Ridiculous as that sounds, some applications will actually interpret the ambiguous expression in exactly that way. Always play it safe and use parentheses to force the interpretation (see Section 11.2 if you need a refresher).

Structural Complexity Within a Comparison.

Some comparisons seem to require more complexity within a single cell. Some actions may require two tests. For example, suppose there were no standard operator "less than or equal." You could build the equivalent by saying:

*If first < second
then display answer 1
otherwise if first = second*

*then display answer 1
otherwise display answer 2.*

The equivalent line in a spreadsheet looks like:

=IF (A3 < A4, D1, IF (A3 = A4, D1, D2))

Just as one step of an algorithm can be any well-defined action, the action part of a spreadsheet conditional can be any well-defined action. In this case it is another call on the conditional function. Chapter 18 addresses such complex functions.

Exercises

- 17.11 Example 17.2 did not account for the analogous problem involving minutes and hours, as would occur, for example, if the experiment started at 3:50:17 and ended at 4:03:28. Build a spreadsheet that also accounts for the minutes.
- 17.12 The representation of a series of dates described in Section 15.3.3 could not deal with the occasional change in year. A conditional such as that used in Example 17.2 can fix that problem. Write a series of dates that includes one day each month for at least two years.
- 17.13 Build a “pocket calculator.” The calculator should use five cells. The user will enter an arithmetic expression into the first three cells. For example, each of the three separate parts of the input “ $3 + 4$ ” should go into a separate cell: “3”, “+”, and “4”). The last two cells display the answer. The fourth cell just contains the constant “=” and the fifth cell contains the value of the expression.
- 17.14 Write a function that determines whether or not the value in one cell is between the values in two other cells.
- 17.15 Write a function that finds the maximum value out of four other cells. You may not use the built-in function `max` (except to check your answer).
- 17.16 What will the following line print out in your spreadsheet?

$= \text{IF} \ (2 * 3 > 5, \ 6 > 5, \ 6 < 5)$

[Warning: this is not the same for all spreadsheets and almost certainly not what you would expect.]

17.5 Summary

Conditional actions are performed (or not performed) on the basis of a test. The test must be a well-defined comparison of objects. In the case of spreadsheets that means cells (or mathematical functions of cells). The actions can be any well-defined action. In the case of spreadsheets that means any action that could be placed in a cell by itself (including another conditional).

Important Ideas

logic	conditional	true
	action	false
test	action	IF
length		

18

Applied Logic

Logic is the anatomy of thought.

JOHN LOCKE

18.0

Overview

Many decisions cannot be reduced to a single true/false question. Fortunately, a great many of these can be reduced to a small set of true/false questions. Boolean logic allows us to represent combinations of true/false questions to produce representations for questions with either arbitrarily complex conditions or arbitrarily complex answers. Boolean logic provides a basis for manipulating those representations.

18.1

Boolean Logic

The test portion of conditional actions (as described in Chapter 17) is restricted to questions with a simple yes/no answer. This seems insufficient for many problems. For example, suppose you are purchasing a present. You feel that anything under ten dollars would be considered cheap or tacky. On the other hand you know you can't afford more than \$20. In selecting a gift, you really need to ask two questions:

Is the price greater than \$10?

and

Is the price less than \$20?

Boolean logic,¹ is a system for combining two such values, and is exactly the tool needed to ask such compound questions. Although a very simple system, it is one of the fundamental tools of all

computer science. No matter what an application appears to do on the surface or how a computer appears to be built, underneath it uses a Boolean representation for essentially every operation, from holding the

1. Named after the English mathematician George Boole (1815-1864) who developed the system for manipulating logical expressions known as Boolean algebra.

value of a variable to performing complex arithmetic. While the new user does not need to analyze complex machine operations, a quick introduction to logic provides a valuable basis for expanding the usefulness of conditional actions.

18.1.1

Conjunction (And)

Recall that the test in a conditional can be any question that can be answered true or false. The two questions above can be restated as a single true/false question:

Is the answer to both of the following questions true?:

Is the price above \$10?

and

Is the price below \$20?

The logical operator *and*, does exactly that: it combines two questions into one question. A question of the general form:

question1 And question2

is `true` if both questions (*question1* and *question2*) are `true`, taken individually. It is `false` in all other cases: if *question1* is `false`; if *question2* is `false`; or if they are both `false`. The behavior of the *And* operation can be summarized as a table of results called a *truth table*:

first argument	second argument	result of first And second
<code>false</code>	<code>false</code>	<code>false</code>
<code>false</code>	<code>true</code>	<code>false</code>
<code>true</code>	<code>false</code>	<code>false</code>
<code>true</code>	<code>true</code>	<code>true</code>

Notice that the first two columns contain every possible combination of two `true/false` values. The third column shows

the result of combining two questions with the And operator. Thus, if the answer to each of the questions:

Is the price above \$10?

and

Is the price below \$20?

taken individually is true, then the last line of the table shows that

(Is the price above \$10?) And (Is the price below \$20?)

is also true. If a price is in the correct range, both answers will be true and the value of the conjunction is also true. But if the answer to either question is false (for example if the price is too low or if it is too high) then the conjunction is false.

When referring to the logical operator `And` in an English sentence, the sentence is usually less confusing if you use the formal name, *conjunction*. For example:

Conjunction is a Boolean operator.

seems more understandable than

And is a Boolean operator.

For that reason the formal terms are used throughout the remainder of this chapter.

Conjunction in a Spreadsheet.

Every spreadsheet provides a built-in function `and` exactly equivalent to the logical operation `and`. Infix notation (see Section 9.4.2) seems ideally suited for representing conjunctions: they are a binary operation, and there are several standard and concise representations for conjunction:

`and`: *question1 and*

question2

`^` : *question1 ^*

question2

`:` *question1 question2*

Unfortunately, most spreadsheets use the prefix notation usually used for the so called built-in functions:

`and` (*question1, question2*)

In the example question, if D3 contains the price, and E1 and F1 hold the \$10 and \$20 cutoffs respectively, the above question is:

`And ((D3 > E1), (D3 < F1))`

which, in turn, can be nested in a conditional statement:

```
IF (And ((D3 > E1), (D3 < F1)), "Possible", "Nope")
```

Notice that Boolean expressions can easily fill up with nested parentheses: the conditional itself uses one set, the conjunction operator needs a set, the individual comparisons may contain parentheses for clarity, and the arithmetic expressions will need parentheses unless they are the simplest of expressions. Take care to match the left and right parenthesis of each pair correctly.

18.1.2

Disjunction (Or)

Conjunction enables the user to combine two questions, each of which restricts the conditions under which the first action is performed. Tests can also be combined to allow alternative conditions for the first result:

Is one condition true?

or

Is another condition true?

For example, the criteria for admission to a campus event might be:

Is the person a student?

or

Does the person have a university activity card?

This is equivalent to:

Is the answer to either of the following questions true?:

Is the person a student?

Does the person have a university activity card?

Notice that this is still a true/false question. The answer is not “a student” as it might have appeared in the original question.²

Questions of the form:

question1 or question2

are called *disjunctions*. Notice also that the disjunction is an *inclusive disjunction*: it returns `true` if the first alternative is true, or if the second alternative is true and even if both are true. This contrasts slightly with common English usage. For example, to most students, the question, “Do you want to study or go to the party?” implies that you expect them to do one or the other, but not both. The inclusive `Or`,³ as used in computer science however, allows for the possibility that a student would do both, as summarized in the following truth table.

first argument	second argument	combination of first Or second
false	false	false
false	true	true
true	false	true
true	true	true

The disjunction, then, might be stated as

Is at least one of the following true?

Is the person a student?

or

Does the person have a University activity card?

2. This treatment of “or” as a yes-no question is a continual source of obnoxious jokes and responses from computer scientists. For example when asked, “Would you like chicken or roast beef for dinner?” we are all too likely to answer ”yes,” rather than a more informative “chicken.”
3. Another form of “or” used in computer science: the *exclusive or* is roughly equivalent to the common English usage. The *exclusive or* has its own important uses in computer science, and its own symbol, but it is not particularly important here.

Disjunction, like conjunction, seems well suited for infix notation: it has several standard and concise representations:

`or` : *question1* or
 question2

`v` : *question1* v
 question2

`+` : *question1* +
 question2

Again, unfortunately, most spreadsheets use the prefix notation:

`or (question1, question2)`

The above question appears as:

`=if (Or ((D3 = "student"), (D3 = "cardholder"))), "admit"`

which places a flag (e.g., “do not admit”) in the appropriate cell. The parentheses distinguish the individual items. (For a review of the disambiguation, see Section 11.2.1.)

Disjunction and Numeric Expressions.

The examples above miss an important case: values that fall exactly between two conditions. In the purchasing example, for instance, rejected \$20. How can one include the question, “Are you old enough to purchase a drink?” In most states, one is considered old if one is at least 21 years old, including exactly 21. The question could be phrased:

`(age > 21) Or (age = 21)`

Compare this with the numeric relation:

`age ≥ 21`

discussed in Section 17.1.2.

Exercises

18.1 Using a spreadsheet, build the truth tables for conjunction and disjunction. Do not simply write the values in the result column, but use the spreadsheet to calculate them.

18.2 Using a spreadsheet, build a truth table that demonstrates the claim that “ $A > = B$ ” is exactly equivalent to $(A > B)$ Or $(A = B)$. It should contain values of A and B in all possible relations to each other. Do the same for “ $A < = B$ ” and $(A < B)$ Or $(A = B)$.

18.3 A truth table showing a combination of two arguments has $4 = 2^2$ rows. How many rows must a table representing a function of three arguments have?

18.1.3

Negation or Not

One more logical operator, `Not`, completes the basic set. `Not` is just the logical inverse, or *negation*, of an expression. Negation provides a way of asking:

*Is the following question false?
any question*

For example:

Are you a non-student?

is roughly

*Is it false that
You are a student?*

If an expression is `true`, its negation is `false`; if the expression is `false`, its negation is `true`, as shown by the truth table:

		Not
		expression
expression		
	false	true
	true	false

Again, most spreadsheets use the prefix notation for negation:

`Not (question)`

The above question can be represented in a spreadsheet as:

```
=if (Not ((D3 = "student")), "false", "true")
```

Infix notation was not really an option for negation: it is a *unary* (only one operand, not a binary operation). Negation does have several standard and concise representations:

`not` : `not`

question

¬ : \neg *question*

— : $\overline{}$ *question*

~ : \sim *question*

overbar: *question*

but most spreadsheets do not use them.

Negation could be used in place of the combination arithmetic operands such as “greater than or equal.” The expression “greater than or equal” includes all cases except “less than.” Thus

$A \geq B$

is the negation of:

$A < B$

If A is less than B , then it is not greater than or equal to B , and vice versa.

Exercises

18.4 Demonstrate with a spreadsheet that $A \geq B$ is actually the negation of $\text{Not } (A < B)$.

18.5 Demonstrate with a spreadsheet that $A \leq B$ is actually the negation of $\text{Not } (A > B)$.

18.1.4

Boolean Operations as Functions

Notice that the Boolean operators are actually defined as functions from logical values to logical values. Stated in the functional notation of earlier chapters, conjunction is:

And: $\text{Boolean} \times \text{Boolean} \rightarrow \text{Boolean}$

That is, conjunction maps a pair of Boolean values (each of which may only be `true` or `false`) into a single Boolean value. At first, this may seem odd. In the case of the Boolean functions, the arguments of the function are usually themselves functions or relations. That is, the expression $(a = b)$ is itself a function from numbers \times numbers to Boolean values. The disjunction

$(a = b) \text{ or } (a < b)$

maps the results of the two arithmetic comparisons into a single value true or false. But, unlike the arithmetic functions such as \times or $+$, Boolean functions have very few elements and can therefore be displayed as a simple table. Such reemergence of concepts from one

area of computer science into other areas provides much of the continuity of the discipline. Recognition of such repeated concepts makes the discipline easier to understand.

18.1.5

Boolean Operations and Nonnumeric Arguments

The Boolean operators are defined for any true/false questions. That means they could also be used to combine the results of comparisons of objects such as strings and months. For example, if D3 and D4 contain strings, the comparison:

IF (Not (D3 = D4), "not the same", "exactly the same"

prints a warning if the strings are not identical. If the two cells contain comparison

```
IF (AND (MONTH (D3) = MONTH (D4), YEAR (D3) = YEAR (D4)), "SOME TEXT")
```

checks to see if the dates are in the same month of the same year. Again, you must force the order of evaluation with careful use of parentheses.

Exercises

18.6 Check to see if the length of one string is greater than or equal to that of a second string. Solve this by two distinct methods.

18.7 Check to see which of two dates comes earlier in the month.

Check to see which is in an earlier month.

18.2

Boolean Algebra

There are questions which cannot be phrased with a single Boolean expression. Suppose a fund raiser had a database of big contributors, and that she was trying to find out if someone whose first name is “George” and whose last name is either “Hyde” or “Hydde”. Notice that this question would involve both a conjunction and a disjunction. Fortunately, the AND and OR operators can be applied to the results of another Boolean operation. Stated formally, the question is:

```
(first name = "George")
```

and

```
((last name = "Hyde") or (last name = "Hydde"))
```

Notice again the use of parentheses to distinguish this question from

```
((first name = "George") and (last name = "Hyde"))
```

or

```
(last name = "Hydde")
```

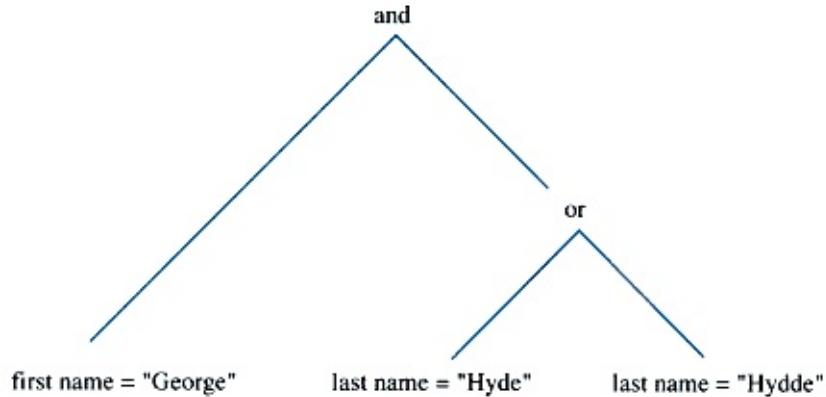
which seems to match either someone with the name “George Hyde” or someone with the name “Hydde”. Notice that the desired query seems to be a conjunction of the two possibilities.

questions, the second of which is itself a disjunction. For use in a query the query can be translated to the appropriate prefix notation as:

```
= And (  
    (first name = "George"),  
    Or ((last name = "Hyde"), (last name = "Hydde"))
```

In general, expressions can be arbitrarily complex. It may be easiest

the question as a tree of logical operations where the tree structure corresponds to the parentheses.



The rules for the construction of complicated expressions from simpler ones (or the breaking down of complicated ones into simpler ones) are called an *algebra* in this case *Boolean algebra*. Contrast Boolean algebra with the algebra you learned in high school: both are systems for manipulating symbols in order to evaluate expressions. However, Boolean algebra has only two possible values and really only three basic operators. In practice, the highest level construct (e.g., the root of the tree) is built from intermediate level constructs, which in turn are built from low level constructs (e.g., the leaves).

18.2.1 Complex Truth Tables

A complex function can also be represented as a truth table, which can be built from the previous truth tables:

first = George last = Hyde last = Hydde Hyde or Hydde George & either

true	true	true	true	true
false	true	true	true	false
true	false	true	true	true
false	false	true	true	false
true	true	false	true	true
false	true	false	true	false
true	false	false	false	false

false false false false false

Notice that the truth table now requires eight rows to account for all possible tests. The third line shows the result of looking for either last name. The last column is the conjunction of that result with the search for the first name.

18.2.2

Manipulating Boolean Expressions

Boolean algebra actually refers not only to the rules for combining simple comparisons into more complex comparisons, but to a set of rules for manipulating Boolean expressions to create new and equivalent expressions. For example the simplest rule, the rule of *double negation* is:

$$\text{Not}(\text{Not } A) \gg A$$

meaning “an expression containing a double negation can always be replaced by the expression with the negations removed” (notice that this rule reflects the rule of English grammar that says “never use a double negative”). A truth table can show the *validity* of this expression:

Double negation

<i>A</i>	<i>not A</i>	<i>not not A</i>
true	false	true
false	true	false

The values in the third column, derived as the negation of the second, are identical to the first column. Therefore, the two expressions will always have the same value, and can be substituted freely for each other. Although a thorough treatment of the rules of Boolean algebra is beyond the scope of this text (it can be found in any introductory logic course), familiarity with some of the basic rules can help you simplify your logical expressions. Proof of these rules is left as an exercise.

Distributive laws

$$\begin{aligned} \text{Not}(A) \text{ And } (\text{Not } B) &\gg \text{Not}(A \text{ Or } B) \\ \text{Not}(A) \text{ Or } (\text{Not } B) &\gg \text{Not}(A \text{ And } B) \end{aligned}$$

De Morgan's law4

$(A \text{ Or } B) \text{ And } (A \text{ Or } C) \Rightarrow (A \text{ Or } (B \text{ And } C))$

$(A \text{ And } B) \text{ Or } (A \text{ And } C) \Rightarrow (A \text{ And } (B \text{ Or } C))$

Vacuous cases

$(A \text{ Or } \text{True}) \Rightarrow \text{True}$

$(A \text{ And } \text{True}) \Rightarrow A$

4. After Augustus De Morgan (1806-1871), a British mathematician and logician who helped lay the foundations for modern logic. Notice that he was a contemporary of George Boole.

In each case A , B , and C may be any Boolean expression. Notice that one or more of the above laws actually apply to almost every Boolean expression.

Exercises

- 18.8 For each of the two versions of the distributive law, write Boolean expressions for your spreadsheet equivalent to each side of the law. Use the spreadsheet to verify both versions of the law.
- 18.9 Repeat Exercise 18.8 for De Morgan's law.
- 18.10 For each of the two versions of the distributive law, build a truth table to verify the law.
- 18.11 Repeat Exercise 18.10 for De Morgan's law.

18.3

Questions with More Than Two Possible Answers

Suppose an instructor wants to automate the assigning of final grades. If the school uses the pass-fail system, the actual grading algorithm is equivalent to a true/false question. Once the individual totals have been calculated:

Calculate student's grade as:

If total score < cutoff

then grade = "Fail"

otherwise grade = "Pass".

However, if the school uses a more complicated system, the algorithm also becomes more complex. The traditional "A, B, C, D, F" system must allow five possible answers. It is not a simple yes/no question. No matter how deeply logical operators are nested in the Boolean expressions, the ultimate answer was still "yes" or "no". Another method is needed. Before attempting the five-grade

example, consider a simpler toy problem: a three-tier system with grades “Great,” “OK,” and “Fail.” The algorithm above works for failures: if the grade is below the cut-off, the student fails. An analogous statement could be used for the “great’s”: if the grade is above the upper cut-off, the student earns a “Great”. But so far there is no way to describe the middle grade, “OK”. The middle group requires something like:

*If score is between first cut-off and second cut-off
then the grade is “OK”
otherwise . . .*

This question is really two questions:

Is the score above the first cut-off?

and

Is the score below the second cut-off?

So the full calculation might be stated something like

Calculate student's grade as:

If total score < cut-off for pass

then grade = "Fail"

If (totalscore > = cut-off for pass) and (total score < cut-off for great)

then grade = "OK"

If total score > = cut-off for great

then grade = "Great"

This “algorithm” has some problems.

1. Algorithms are performed stepwise, but this solution does not seem to recognize that fact. But if any one test is true, the other two should be false. Only one action should ever be performed, but this algorithm always asks each question, even though it can't possibly be true.
2. Although the programmer here was careful to make the tests mutually exclusive, that care may require a major duplication of effort. Notice that each of the two subparts of the middle test essentially duplicates one of the other two tests.
3. Finally, it may not be clear how to put all three tests into a single cell. Although it is possible to use three cells one for each possible grade that would place a student's grade in any one of three columns. That would work, but it is not very elegant: a column of grades for the class would spread across three columns, looking ragged and using up more space on the spreadsheet.

Actually the middle problem helps answer the other two. The tests are in fact mutually exclusive. Therefore if any one test is true, the other two must be false. An algorithm can take advantage of the mutual exclusion:

Calculate student's grade as:

- (1) If $\text{total score} < \text{cut-off for pass}$
- (2) then $\text{grade} = \text{"Fail"}$
- (3) otherwise If $(\text{total score} < \text{cut-off for great})$
- (4) then $\text{grade} = \text{"OK"}$
 $[\text{cut-off for great}^3 \text{ total score}^3 \text{ cut-off for pass}]$
- (5) otherwise $\text{grade} = \text{"Great"}$
 $[\text{total score}^3 \text{ cut-off for great}]$

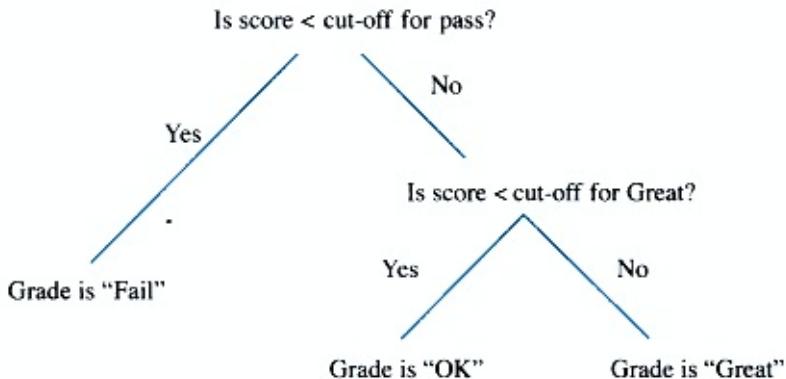
Following the algorithm through: if line (1) is false, then $\text{total score}^3 \text{ cut-off for pass}$, which is half of the criteria for the second test. If the other half is also true (line 3), the grade is “OK”. If it is neither of the first two options, the grade must be “Great.” The *otherwise* action is itself a complex action.

Fortunately, the original definition of conditional action allowed the two actions to be arbitrary. This translates directly into a spreadsheet format in the obvious way: state the complex action in exactly the way that you would have stated the simpler action:

	A	B	C	D	E
1	Cut-offs	for "great":	70	for "ok":	50
3	student name	total score	grade		
4	Ivan	82	great		
5	John	45	fail		
6	Jose	51	ok		
7	Jac	69	ok		

1	Cut-offs	for "great":	70
3	student name	total score	grade
4	Ivan	82	=IF(B4<E\$1,"fail",IF(B4<C\$1,"ok","great"))
5	John	45	=IF(B5<E\$1,"fail",IF(B5<C\$1,"ok","great"))
6	Jose	51	=IF(B6<E\$1,"fail",IF(B6<C\$1,"ok","great"))
7	Jac	69	=IF(B7<E\$1,"fail",IF(B7<C\$1,"ok","great"))

Schematically, the test looks like:



Completion of the full five-grade problem is left as an exercise. Section 18.4.1 discusses an alternative provided by many spreadsheet applications for larger multivalued problems. In general, any number of discrete values may be generated with compound questions. This simple ability to create complex questions from simple Boolean queries is a cornerstone of computer science.

Exercises

18.12 Build a function that tests whether a numeric value is greater than, less than, or equal to zero. Build a spreadsheet using the function to print out the words “greater than”, “less than”, or “equal”.

18.13 Build a function to compute a letter grade for the traditional

A, B, C, D, F system.

(table continued on next page)

(table continued from previous page)

18.14 The eightline truth table in Section 18.2.1 seems to imply that there are three possible true answers for the question. How many are there really? Explain the difference.

18.4 Logic in Applications

The basic logic principles appear repeatedly in any computer application. Consider a few special cases.

18.4.1 Tables and Conditionals

The grading example in the previous section is typical of a large class of problems in which all objects of a given type are divided into several mutually exclusive groups, each group represented by a range of values.⁵ In the grading case, the set of objects is the students and the mutually exclusive groups are the range of grades. In everyday communication, the most common way to represent such ranges is not with a series of conditionals, but with a table:

From (minimum) To (maximum) Grade

0	45	F
46	60	D
61	74	C
75	88	B
89	100	A

Most spreadsheet applications provide a built-in tool, called `table lookup` or `index`, that is much more convenient than a long list of nested conditionals. Roughly an index works as in the following algorithm. A test value is iteratively compared to a table of ranges, and result scores:

Repeat until score is in the appropriate range:

If test value is in the current range

*then find the score in the corresponding column of the
 table*

otherwise look to the next line and

if more lines exist

then repeat the algorithm

5. This is exactly the notion of *disjoint covering* (disjoint = nonoverlapping; covering = including all objects in the set) used in topology and other subfields of mathematics.

6. Note the clash with the definition of algorithm here: this algorithm does not say what to do if no appropriate range is found.

The problem of determining if the test value is in the given range is made easier by listing the ranges in consecutive order. Just as in the previous section, each test can be restated

*If the value is in the current range
then find the corresponding grade
otherwise it must be in one of the later ranges.*

At each step, it is only necessary to test one end point in the range. For example, if the test value is 80, the test for the range 75-88 is reduced to a comparison to 88. Values less than 75 are ruled out by earlier steps. Figure 18.1 shows a typical spreadsheet table with lookup commands. The exact syntax of table lookup commands will vary from application to application as will the available options. For example, you may be able to search from high values to low values, rather than low to high.

18.4.2

Databases and Logical Expressions

Database applications, like spreadsheets, allow users to request arbitrarily complex information by means of Boolean expressions. The “George Hyde” example, above, is just a small example. Consider a more complex example:

Example 18.1 The computer science department at a large university wishes to select the student of the year. They ask the registrar to search for candidates with the following qualifications. The student must: be either a senior or have

B	C	D	E	F
Grade ranges		individuals		
minimum				
0	F		score	grade
45	D	John	80	B
64	C	Mary	92	A
74	B	Bill	40	F
88	A			
100	A+			

B	C	D	E	F
Grade ranges		individuals		
minimum				
0	F		score	grade
45	D	John	80	=LOOKUP(E4,B\$3:B\$7,C\$3:C\$7)
64	C	Mary	92	=LOOKUP(E5,B\$3:B\$8,C\$3:C\$8)
74	B	Bill	40	=LOOKUP(E6,B\$3:B\$7,C\$3:C\$7)
88	A			
100	A+			

Figure 18.1
Calculating Grades via a Table Lookup

been on campus for at least 5 years; be over 20 years old or be married (grade point average of over 3.5 for all courses or 3.65 for CS courses 4321). Any student who has an overall grade point average of 4.0 will be independent of all other qualifications. The following database query candidate students:

```
Or (
  And (
    Or (year = senior, time_on_campus >= 5),
    Or (age > 20, married = yes),
    Or (total_GPA >= 3.5, CS_GPA >= 3.5,
        CS4321_grade = A +)
    (total_gradepoint = 4.0))
```

18.4.3

Manipulation of Relations

In addition to selecting records with complex descriptions such as all grade > 5 or year = senior, logic helps form new relations. Two additional operations, *project* and *join* are also based on logical operation Project.

Although the *select* operator can find entire records matching a complex description, it is not always sufficient for realistic database manipulation. Suppose we were the registrar at a university and were asked to provide a list of all the students' addresses. Unfortunately the student records contain more than just their addresses. They contain information that is more private, such as grades. A user with a legitimate reason for requesting the addresses may not have a legitimate reason for seeing the grades. The *project* operation allows the user to choose certain fields. Where the *select* operation chooses certain rows or columns, the *project* operation chooses certain fields or columns. For example, consider the relation student:

```
student (name, address, major and grade average)
```

the operation *project* might be stated as:

Project student onto [name, address, major]
yielding a three-field relation. The select and project operations
together to produce any subset of a single database. In the current ex

*Project student onto [name, address, major]
[remove the private information]
Select all records with major = CS
Project onto [name, address]
[major is redundant since all selected students]*

7. Unfortunately many student databases do not support these actions.

Join.

In more complete database systems, separate relations can be *joined*—combined—into new relations. For example, the student relation above could be joined with the payment relation:

payment (student name, current balance, status)

to form one large relation:

temp (name, address, major, grade average, name, current balance, status)

Notice that the resulting relation has two fields called *name*. These are the same field—the one that identifies the student. So one of them can be omitted:

temp (name, address, major, grade average, current balance, status)

In fact, the *join* operation requires that the two relations share at least one common field. This requires the user to identify that field:

Join (student & payment) on name.

Finally notice that the *join* relation produces some surprising results. For example, joining the student relation with a faculty relation would very likely produce a null answer. Joining the student relation with the grade relation:

grade (course, name, grade)

would result in the combined relation:

temp (name, address, major, grade average, course, grade)

That looks fine, but notice that this relation has a record for every student who took five courses. If a student took five courses, there would be five records for Bill, each containing information about the course but also the other fields such as his address and major. Joining relations creates redundant records, but it does allow for combining information that would otherwise be inaccessible.

18.4.4

Testing, Debugging, and Representing Complex Logical Structures

In practice, the representation of complex logical questions can be useful. It is so complex but because the representation as a single line of text can be easily understood by a computer.

expression may be too long to fit on the screen or in the open window. The nested parentheses can become confusing, making it difficult to determine what each operator does. The expressions are actually the arguments for each operator. As with arithmetic operators, the extent to which an operation applies is called its *scope*. In the expression `Ors was`, the scope of each of the three consecutive (and most indented) `Ors` was limited to the expression that followed on the same line. The scope of the `And` was limited to the expression that followed on the same line.

the three indented `Ors` (and their arguments). Finally, the scope of the first `Or` was the entire complex expression.

One useful technique for constructing complex Boolean expressions is the use of intermediate values, exactly the same as with arithmetic expressions (for a refresher, see Section 11.2.1). Build each subtest individually. That way you can verify both the correctness of the test and the conditions that you are testing. For example, instead of calculating the single cell:

```
B3 = If (A3 = A2, If (A3 > A5, x, y), If (A2 < A5, w, z)),
```

calculate two intermediate values before calculating the final value:

```
B1 = If (A3 > A5, x, y)
B2 = If (A2 < A5, w, z)
B3 = If (A3 = A2, B1, B2)
```

If you wish, you can recombine the expressions after testing the individual parts.

The presence of complex Boolean expressions probably indicates the possibility of many distinct input conditionseach of which must be tested. Always test representative examples for all possible combination cases.

18.5

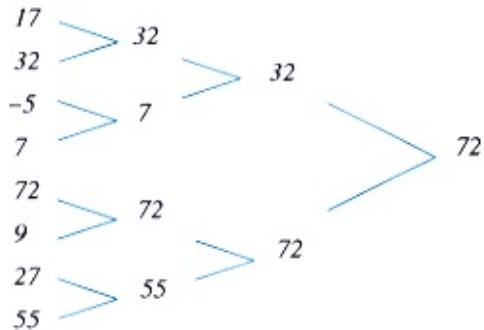
Conditionals and Hierarchical Structures

Conditionals are one of the primary building blocks used by computer scientists for constructing arbitrarily large algorithms. The others are iteration (Chapter 15) and procedures (Chapter 19). The function `maximum` (described in Section 17.3.2) can be defined using multiple conditionals as:

*For every node in the tree
if it is a leaf
then its value is the value in the cell*

otherwise its value is the larger of its two children.

One way to accomplish this is with a tree structure. Visualize the list as the leaves of a tree.



The technique is roughly: Compare each pair of adjacent cells (an odd row and the following even row) to find the larger, using a statement something like:

```
if (B1 > B2, B1, B2)
```

It is only necessary to find this for half the cells in the column, say, the even ones. Repeat the process on the next column and succeeding columns until there are none left.

Why would you want to build your maximum function from scratch, rather than just using the built-in function `Max`? The first reason is that you might want to find the “maximum” of nonnumeric objects, for example: the last item alphabetically, or the latest date. Although most spreadsheet applications allow simple comparisons of nonnumeric objects, the function `Max` is usually restricted to numeric objects. Exercise 18.16 explores this extension.

In addition, algorithms can be reused, even when code cannot. The algorithm for finding the maximum of several numbers is conceptually identical to the algorithm for finding the “maximum” of several character strings. In fact, it is almost identical to a great many algorithms such as finding the minimum or the closest to the average. This technique used here is essentially a *binary search*, a very efficient and common search algorithm. Although a spreadsheet implementation does not capture the real speed advantage of a binary search, it does capture the simplicity with which many large problems can be solved.

Exercises

18.15 Find the minimum value in eight spreadsheet cells by the above method.

18.16 Place character strings into eight cells of a spreadsheet. Then find the alphabetically last of those strings.

18.17 Suppose you have a list of numbers in cells B1 through B16. Find the value that is closest to the average of all the values. Hint: that will be the value whose difference between it and the average is minimum.

B	C	D	E
17			
32	=IF(B1>B2,B1,B2)		
-5			
7	=IF(B3>B4,B3,B4)	=IF(C2>C4,C2,C4)	
72			
9	=IF(B5>B6,B5,B6)		
27			
55	=IF(B7>B8,B7,B8)	=IF(C6>C8,C6,C8)	=IF(D4>D8,D4,D8)

B	C	D	E
17			
32	32		
-5			
7	7	32	
72			
9	72		
27			
55	55	72	72

Figure 18.2
Searching for the Largest

18.6 Summary

Boolean logic and Boolean algebra provide powerful tools for asking arbitrarily complex questions. The complexity from questions can be due to the interaction of the possible initial conditions, or the need for more than two answer cases. The basic Boolean operators are negation (`Not`), disjunction (`Or`) and conjunction (`And`). Truth tables provide a quick and efficient method for checking the values of compound Boolean questions. Both spreadsheets and databases allow users to ask complex questions through the use of Boolean algebra.

Important Ideas

And	Not	Or
disjunction	conjunction	negation
Boolean	logic	truth
algebra		table
unary	binary	valid
project	join	

19

Abstraction, Abbreviation, and Macros

*Miss. A title with which we brand unmarried women to indicate that they are in the market. Miss, Missis (Mrs.) and Mister (Mr.) are the three most distinctly disagreeable words in the language, in sound and sense. Two are corruptions of Mistress, the other of Master. . . If we must have them, let us be consistent and give one to the unmarried man. I venture to suggest *Mush*, abbreviated to *Mh.**

AMBROSE BIERCE¹

Pointers

Gateway Macros

Labs: Advanced Macros

Lab Unit 11: Design of

Manual: Reusable Tools

19.0

Overview

You have already seen procedures and macros: tools that perform whole sets of actions in response to a single request. For example, the built-in functions of spreadsheets are actually procedures. Thus far, all of the procedures you have seen were designed by others. Chapter 19 demonstrates techniques for creating your own tools in the form of procedures or macros, thereby customizing the applications to your own special needs.

1. Bierce lived from 1842-1914. This quote, from *The Devil's Dictionary*, shows that he was well ahead of his time.

19.1

Abstraction

Abstraction is one of the most powerful concepts in computer science. You have seen it repeatedly throughout this text. An outline uses abstraction to capture the essence of each section. Iteration is the repetition of abstract concepts. Applying one set of principles to many applications represents abstraction or generalization of the application. The built-in functions of a spreadsheet are an example of using an abstract idea rather than the individual steps. Two specific toolsabbreviations and macrosenable you to make your abstract ideas explicit. It should not be a surprise that many applications provide a general mechanism for users to create and use their own abstractions.

19.1.1

Abbreviations

An abbreviation is a shorter or more convenient form of a longer expression. The most familiar form of abbreviation is the shortened natural language name:

etc. for *et cetera*

lb for *pound*

USA for *United States of America*

UK for *United Kingdom*

WWI for *World War One*

ID for *identification*

The written representation may be shorter as in *etc.* or *lb*. Some abbreviations, such as *ID* or *UK* are shorter to pronounce.² Some are only shorter in one of those dimensions: *WWI* is shorter to write

but not to pronounce than *World War One*.

Nicknames.

Some abbreviations, such as nicknames, seem to show special attachment. People say *Bev* for Beverly or *John* for Jonathan or *Old Glory* for the flag of the United States, not just because it is shorter but because the nickname indicates a special attachment or familiarity. An abbreviation often is most useful in familiar situations. A nickname is never required, but provides an alternative that may be preferable in some circumstances.

Acronyms.

Multiple word names of many organizations or objects can be shortened into a single new, but pronounceable word by using the first letter of each word:

WHO World Health Organization

laser light amplification by stimulated
emission of radiation

ram random access memory

(table continued on next page)

2. Notice that *United Kingdom* is itself a short form, or abbreviation, for *United Kingdom of Great Britain and Northern Ireland*.

(table continued from previous page)

AIDS acquired immune deficiency syndrome

CDromcompact disk readonly memory

risc reduced instruction set computer

Such new words, called *acronyms*, are a special form of *mnemonic*. Often an acronym is not just shorter, but it is also easier to remember than its fully expanded name. This is particularly true in new and emerging areas such as computer science or biotechnology. Long names of new products and ideas from these disciplines often contain strange words and phrases that lay persons find confusing (and care less about). The acronym is simply easier to remember. In many cases the basic idea is pushed to include combinations formed from sequences other than first letter of each word, such as:

FORTRANFormula translator

Cobol Common business oriented language

sonar sound navigation and ranging

radar radio detecting and ranging

modem modulatordemodulator

Some acronyms condense so much meaning into a short word that

they become standard terms of the language, sometimes so much so that the original phase becomes lost. By some accounts, *posh*, meaning "smart or fashionable," originally, referred to the most desirable cabins on steamships those that were "port outwardbound; starboard homewardbound."

19.1.2

Abbreviations in Computing Systems

In addition to the many acronyms that are so common to the computing industry, many other forms of abbreviation appear in the computing environment. Anything, not just names, may be abbreviated. In particular, instructions or sequences of instructions often have easier-to-use common forms. By this point in the course, most students have found some of the most obvious of these abbreviations. For example, in many systems, the single action *double click* is equivalent to:

Select an object (usually with a single mouse click).

Open the object (usually from the File menu).

The double click is just an abbreviation, a faster way of accomplishing the same thing. Other abbreviations are matters of personal preference or convenience. Most users find menu selection using a *deictic*,³ or pointing, device, such as a mouse, amply convenient. However, many users also find that for at least some

³.*Deictic* means pointing out or demonstrating directly. In computing systems, a mouse is the most common tool for deictic input: it points to the desired item. Other deictic input devices include a *track ball*, touch sensitive screens, and a *stylus*.

circumstances, pointing is not the most convenient way to select an action. Any special action that is performed as part of the process of typing may create inconvenient hand movements. For example, the creator of a document containing many bold text segments may find the process required for making individual segments bold inconvenient. Every instance of bold text requires a sequence of actions, such as:

Move the hand from the keyboard to mouse.

Use the mouse to select a menu commands

(e.g., Bold from a Font or Style menu)

Move the hand back to the keyboard.

which sets the style for the following characters to bold. A similar series of actions is needed to revert to the standard text, or to make a previously written segment bold. For this reason, most systems and applications provide abbreviations or alternatives for commonly used menu items. Many of these abbreviations are especially convenient because they are standard across many (or all) applications. For example, the `Quit` or `Exit` command can be abbreviated as `COMMAND4Q` (the command and Q keys pressed simultaneously) on the Macintosh, or as `ALTF`, `×5` in Windows systems. Such abbreviations simply substitute one form of command (keystroke combinations) for another (deictic reference with the mouse). Figure 19.1 illustrates a few more common abbreviations, and Exercises 19.4 and 19.5 ask you to find some more for yourself.

Just as some people always write out the words *et cetera* and others always abbreviate it as *etc.*, some users never use these abbreviations and are quite happy that way; others swear they could not live without abbreviations. An abbreviation provides no new computing power just an alternative method of selecting commands. If they are convenient for you, great; if not, then just use the old method.

The designers of most applications use two conventions that make learning abbreviations very easy for the user. First, they display the abbreviation in the menu. Figure 19.1 shows a segment of Edit menus (one from a Macintosh and one from a Windows application). Many commands include an abbreviation, such as a command character and a letter, either next to the command or as an underlined character. Any time that the menu can be used, the user can substitute the command-letter combination. Many users find that for frequently used commands such as those in the edit menu, they use the keystroke abbreviations, and that for less frequently used (and therefore not so well remembered) commands, they use

4. Command is a control key. In a Macintosh, the command key is labeled as: ⌘. Thus, COMMAND-Q is the ⌘ and the Q-key. Also note that command is distinct from control in most applications.
5. The ALT and F keys pressed simultaneously, followed by the x key.

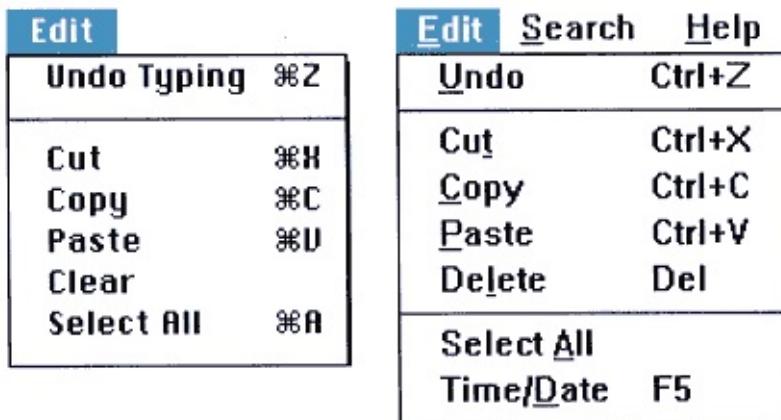


Figure 19.1
Abbreviations Displayed in a Menu

the mouse command. Remember: abbreviations are only of value if they save you time or effort.

Exercises

- 19.1 Write down ten English words or phrases and their abbreviations. For each, indicate: is the abbreviation actually shorter in the written form, or in the spoken form? Do you use the abbreviation yourself?
- 19.2 Cut and paste a text segment using command-key abbreviations.
- 19.3 Select another command and try it.
- 19.4 Find and try the abbreviations for three system level commands on your machine.
- 19.5 Find and try the abbreviations for three commands in your word processor and three commands in your spreadsheet.

In a Mail System.

A *nickname* or abbreviation is useful as a shorter name for persons to whom you frequently send Email. For example,

Bill

might serve as an abbreviation for

President@whitehouse.gov

After defining the nickname, Bill, the user can use it in place of the full address,

President@whitehouse.gov

Box 19.1 A Technique for Learning Abbreviations

Most people eventually conclude that at least some abbreviations would be useful. Unfortunately it may be difficult to remember the abbreviations (that is, after all, one reason for the popularity of graphical user interfaces and menus). When that time comes for you, try the following technique for learning the abbreviation:

*When you start to use a familiar menu item
Find the menu item with a mouse as usual, but
do not actually select it.*

Instead,

*notice the abbreviation
leave the menu, and
enter the abbreviation from the keyboard.*

Taking the time to actually use the abbreviation will help you remember it far better than if you just try to notice what it is and remember it for next time.

to send a message to the president. Most mail systems provide a tool for creating and keeping nicknames, typically a straightforward command such as “make nickname.” The actual steps for creating a nickname include:

*State the nickname.
Give the definition for the nickname.*

Nicknames are interesting for two additional reasons. First, a nickname changes the *interface*, or set of commands that the system will accept. A user who creates a nickname changes the form of future interactions with the machine. The keystroke abbreviations for mouse actions are defined by the system designer; they are constant. But nicknames are created by the user, who now has a new option for interacting with the machine. Second, even the name *nickname* is itself a nickname for the more formal term *alias*.

Alias in a Computing System.

An alias is an alternate name or abbreviation for a document or application in a computing system. From the computing system's perspective, an alias is a pointer to the actual document or application. Selecting the alias is equivalent to selecting the actual document or application. Although some new users find the thought of creating extra names rather astonishing, aliases can be valuable tools for organizing your data files. Aliases are

useful not so much because they are shorter, but because they may suggest the use of the applications or documents. For example, you just purchased a new application called *Performance*, a name that provides little clue that it is a spreadsheet. You could create an alias, say *MySpreadSheet*, that points to *Performance*. In the future you will find the new nickname nice and recognizable. Multiple aliases can enable multiple organizations. For example, many users like to place all files related to an application together in a single folder. For example, a mail system and all of the files that are closely associated with it (e.g., the mail system itself, the incoming messages, and the outgoing messages) might go into a single folder called *Mail Folder*. Then all of these application folders could be grouped together into another folder called something like *My Applications*. This provides a nice tidy and consistent organization but unfortunately, it makes access quite difficult. To start the mail application, the user must first open *My Applications*, then open *Mail Folder*, and finally start the mail program itself. As an alternative, the user can create an alias, say *My Mail* and place it at the highest (e.g., desktop) level. This provides the best of both worlds: the application itself is stored in a logical manner but it is still easy to access.

Exercises

19.6 Although almost universally available, the method of creating nicknames for electronic mail systems varies considerably from system to system. The method for your system is described in Section 11.2.1 of the lab manual. Build and use a nickname for the instructor of this course.

19.7 System aliases also vary widely. They are described in Section 11.2.2 of the lab manual. Build and use an alias for your word processor. Also build one for an individual document that you use frequently. Place the alias in a

different folder than the original.

19.2

Abstraction and Procedures

In most examples thus far, an abbreviation has referred to the name of a single object: a country, person, document, or application. Abbreviations can also represent entire actions or series of actions. Such an abbreviation serves to group a collection of individual steps into a single whole.

19.2.1

Built-in Procedures

In fact, you have already seen abbreviations for groups or series of actions in the form of the built-in functions in a spreadsheet (see Section 9.4). Most built-in functions are actually algorithms specifying a complex action in terms of simpler ones. For example, the function `sum` is an abbreviation for:

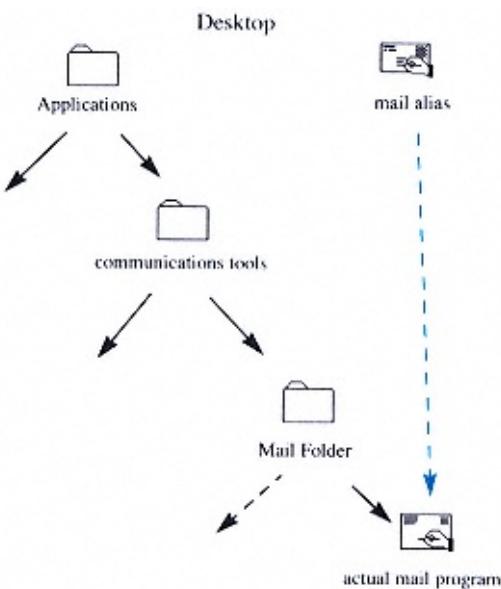


Figure 19.2

An Alias for Easier Access to a Nested Application

Source: The mail symbols used in this figure are actually the icons for the *Eudora* mail system, and for *Applemail*. *Eudora* was created by Steve Dorner and is now copyrighted by Qualcomm, Inc. *Applemail* is copyrighted by Apple Computer.

*For each of the selected items
add that item to the others*

And count is just that

Count the selected items.

Average is an abbreviation for the combination of those two algorithms:

*Count all of the selected items.
Add up the selected items.
Divide the sum by the number of items.*

Abs (absolute value) represents the conditional action:

If argument > 0

*then return the argument itself
otherwise return the negative of the argument.*

The user could specify any of these explicitly step by step, but a built-in function is usually much easier to use. Each built-in function is actually a collection of steps grouped together with a single name. The name is suggestive of the steps taken together much easier to understand than the steps taken individually. From the user's perspective, it appears to be a single action. From the machine's perspective it is all of the constituent parts. In some sense you can think of a built-in function as a pre-made product like clothing or restaurant food. The user (purchaser) need only specify the item; the manufacturer does the individual steps. Again it is the best of both worlds: the user can accomplish complex tasks but need only worry about the surface details.

Such built-in functions have several advantages beyond the obvious. The user not only does less work, but does less "reinventing the wheel." Someone else has figured out the standard or common steps, which frees the user to work on those parts unique to the current problem. Because the built-in function is standard and available to many users, it is well tested. The more use a procedure has had, the less likely it is that it contains an error. Most built-in functions are also quite efficient at least as efficient as any the user is likely to create.

19.2.2

Grouping Actions

Actions other than computations also commonly occur in groups or combinations. For example, in this book, most text is set in the Times font. In contrast, product names are set as *italicized Helvetica* font. For each occurrence of a product name, any author following this convention must perform three separate actions:

1. Change the font (from Times to Helvetica).
2. Change the size (from 12 to 10).

3. Change to italics (from regular).

And of course the steps need to be reversed or undone before continuing with the remainder of a sentence.

A *macro*⁶ is a single command that performs multiple steps, in this case, all three font/style change steps. The user can accomplish the same result by selecting a macro called, say, *product name* that performs all three actions as if they were a single action. More significantly, any user can create a new macro at any time. Most systems and most applications provide both predefined or built-in macros and a mechanism for users to create their own.

6. This chapter refers to several constructs, such as subroutines and macros. Although these two are technically distinct from each other, I will not distinguish between them here. First, the difference is in the internal implementation of the structure. An understanding of that difference is beyond the scope of this text. Second, many user applications actually use the names incorrectly. In such situations, an explanation of the difference would only serve to confuse rather than help.

19.2.3

Predefined Combinations

Macros provide easier ways of accomplishing common actions. You already use several commands that can be thought of as macros. That is, they are single commands whose results can also be obtained by combining other commands. If the command did not exist a user could create it. For example, suppose there were a word processor with no Copy command, just Cut and Paste. A functional (but perhaps impractical) word processor does not actually need the Copy command. To copy a segment and place the copy at a new location, the user could:

Cut the segment.

Paste it right back.

(At this point the user has duplicated the effect of Copy.)

Select a new location.

Paste the segment there too.

Clearly the first two steps are equivalent to Copy. Thus Copy could be thought of as an abbreviation for those two steps.

Similarly, the results of Save-as could be achieved by:

Quit the application.

Duplicate the document.

Reopen the application using the new version.

Fortunately, you do not have to create these actions. In addition, the actions do not work the way the examples suggest they do. What is important here is that the proposed macro produces the same result as the original action.

19.2.4

Other Examples

Many user applications allow the user to build new macros. User-created macros have the effect of custom tailoring the application to

the user's special purposes. Any combination of actions that the user performs frequently can be a macro. Similarly, a user may wish to make a macro for the combination that is used only occasionally, but that is very complex. Reusing a complex process only a few times can save effort. Once the user creates and tests the macro, she can use it with confidence. Some simple examples of potential macros include:

In a word processor:

Make multiple font and style changes concurrently.

Make all of the needed changes (margins, tabs, spacing, etc.) for a new kind of paragraph.

Scroll ahead by two screenfuls.

In a drawing or painting tool:

Select all objects and align them, both horizontally and vertically (e.g., center them).

For a selected object, make its border blue and its fill color red.

Change a line's width and color, and make it into an arrow.

In a spreadsheet:

Add up three specific numbers (but see the caution in Box 19.2).

Create a chart of a specific type and size.

Select a new cell exactly 10 positions below the current one.

Set the style or font for the selected cell.

19.3

Building New Macros

Although the details will vary between user applications, in most cases the process for creating a macro is both simple and consistent:

Turn on a “recorder.”

Perform the action once.

Turn off the recorder.

Name and document the macro.

Save the results.

Making a macro in a graphical environment is much like making a tape recording. The only difference is the items recorded are not sounds but the user's keystrokes and mouse movements. Once recorded, the user can request that the recorded steps be replayed, usually by requesting the macro by name from a menu or by a keystroke combination (abbreviation). Laboratory Unit 11 includes a more detailed discussion of the steps for your applications.

For example, in the above font example, the user would:

Select “the macro creator” tool from a menu.

Push the “start” button in the macro creator dialog box.

Build the macro itself:

Select Helvetica from the Font menu.

Select italics from the Style menu.

Select 10 from the Size menu.

Push the “stop” button in the macro creator dialog box.

Fill in the comments window as:

“Set current font to be 10 point Helvetica for use as a product name.”

This specialty font will be called ‘Product’.”

Fill in the name “Product name” for the macro.

Optional: provide a keystroke abbreviation for the macro.

Save the macro.

Test the result.

To use the Product name macro, the user simply:

Selects the “macro player” tool.

Selects the Product name macro.

Box 19.2 A Common Problem with Macros, or “A Word to the Wise”

In many user applications, macros seem to fall a bit short of what users desire. The problem is that the steps of the macro must be identical every time they are used. For example, a spreadsheet user may want to build a macro that calculates the range or spread of a set of numbers:

Find the minimum of the selected cells.

Find the maximum of the selected cells.

Find the difference

by subtracting the minimum from the maximum.

Unfortunately, many spreadsheet applications do not allow the user to do this in a useful way. The reason is our old friend: absolute addressing. The macro records the exact movements of the user. Thus, if the user selects 10 cells (say D1–D10) with the mouse, those are the exact 10 cells that the macro will select every time it is used. There is no way to use the macro to select the cells F1–F10. This seems unfortunate. Indeed, the built-in functions seem to be able to handle this situation. We will visit macros again in Section 25.4, where we will see a new a tool, called *parameters*, enabling exactly this capability: specifying the objects of interest to the macro or procedure at the time the macro is used rather than at the time it is defined.

19.3.1

Variations

The user can give the macro a name and specify the keystrokes. That is, the keystroke abbreviation is an abbreviation for the macro, which itself is an abbreviation. Some systems even allow the user to create an icon to activate the macro. Macros can run at full speed or at the same speed as the user recorded the original. Generally you

would want a macro to run as fast as possible, but the slower speed might be useful if you wanted to demonstrate the actions to another person.

Some macro recorders create written descriptions of the actions. These are written in a scripting or command language showing the step-by-step listing of the actions. The macro player then reads this language rather than the mouse

command sequences. The advantage of such a listing is that the user can edit it. If you make a small error, just edit the one step rather than rerecording the entire macro. In some systems (e.g., UNIX and DOS), the user actually creates the original command language sequence explicitly by typing the individual commands. The two methods are essentially equivalent. Although command languages are more flexible, they require a higher level of knowledge on the part of the user.

19.3.2

Templates

A *template* is a pattern or blank form used as a model for new creations. In computer applications, they are “dummy” documents containing formatting or other general information about the form of a document. For example, a general writing template can contain the default or initial paragraph (width, spacing, tabs, and so on) and character (font, style and size) settings. Rather than setting up each new document individually, the user can use the template as a starting point, saving the new document with an appropriate name. A more complex template might contain *headers* (e.g., a date), *footers* (like a header but at the bottom of a page, perhaps containing a page number), *signature* (in a letter template), *boilerplate* text (segments of text to be pasted into a document), or *borders* in a spreadsheet (e.g., a user could reuse a mortgage calculation spreadsheet). In many applications, a template can even contain the definitions of macros. A template itself can even be thought of as a type of macro: use of a template is a shortcut for performing all of the actions that were needed to create the original.

Many applications provide explicit tools for creating templates (also called *stationery* or *forms*). But even if there is no specific mechanism for their use, the user can always create a template. Just create a document containing all of the needed definitions and save it. Use this document as a starting point. A word to the wise: when

creating your own templates, you will do well to protect the documents against change (e.g., *write protect* the document or make it *read-only*). That way you will not destroy the template as soon as you create a new document.

19.3.3

Scope

A macro or procedure is only known within the context, or *scope*, in which it is defined (see Section 18.4.4 for a similar use of *scope*). For example, a spreadsheet macro may only make sense within the context of a spreadsheet, so its scope should be limited to spreadsheets. In practice, the scope of a macro is likely to be the document in which it is originally defined. Most applications allow the designer to expand the scope to include similar documents, say all spreadsheets. Once defined for a larger scope, the designer may reuse it in new documents. Many applications allow the designer to use templates to extend the scope to all documents of a similar form.

Exercises

- 19.8 Create at least one of the macros from each of the first three application groups listed above.
- 19.9 For each of the first three applications above, create a macro that does not appear in the above list.
- 19.10 For each kind of application, think of two more potentially useful macros.

19.4

Macros and Abstractions

19.4.1

Named Actions

As valuable as macros are for saving steps, their true value comes from their ability to group actions and give a name to that group. For example, consider the example macro: `Product name`. This abstraction serves many purposes:

1. Every use is consistent. With a macro, there is no danger of sometimes omitting one of the three changes.
2. The user need not remember exactly how to perform the individual steps.
3. The user need not waste time and energy remembering what product names should look like (what font details).
4. Macros are yet another example of the computer science dictum “build reusable tools.”

The user can give a name to the entire group. The name `product name` is far more suggestive than the name “10 point italicized Helvetica.” Because abbreviations and macros emphasize a term as a familiar and well-known concept, they support the abstract

concept. The user does not need to know how a macro works just what it does.

19.4.2

Documentation

Naming Conventions.

Generally the name of the macro should suggest its purpose or the reason for its existence, rather than the steps of how it works. For example, the macro `Product name`, above, describes when to use the macro rather than the individual steps of selecting the font, style, and size of the text. Names selected by purpose are easier to find at a later date because they suggest the reason for needing the macro long after the user may have forgotten what the steps were. In addition, a macro named according to its purpose can be changed easily. If at a later date the user discovers that `product name` should be 12 points high rather than 10, it is relatively easy to change the definition without changing the name. If the name changes, then every instance of the macro will have to change.

Internal Documentation.

A good macro needs two forms of documentation:
what it does, and
how it does it.

This distinction lies at the heart of the macro's usefulness. Once a designer has created a macro or procedure, she need not pay any attention to how it works. The only essential information is what it does. This suggests that information about what a macro does should be easily accessible, but information about how it works can be relatively hidden.

Procedure and function documentation should describe what the routine does, and how it does it (if it is not clear from the body of the routine). Pre- and post-conditions make excellent documentation for procedures.

A Caveat:

Unfortunately, few end-user applications provide very good tools for documenting user-created macros. That makes user documentation all the more critical. Use all the tools available. In particular, name them well. If you create many macros, you may want to create a document that lists them all. (Yes, even if you name them well, you will forget!)

19.5

Summary

Macros are an excellent tool for creating procedural abstraction. A macro or procedure is actually an abbreviation for a longer or more complex series of commands. Various forms of built-in or user-designed abbreviations include procedures, macros, templates, built-in functions, nicknames, and aliases. Although most user applications provide tools for creating your own abbreviations of

macros, unfortunately the tools commonly available to students are often not very powerful. Fortunately, Chapter 24 will reintroduce macros in a more powerful form.

Important Ideas

procedure	macro	function
template	abbreviation	nickname
alias	recorder	abstraction
write protect	read only	boilerplate

20

Visualization Revisited

If history repeats itself, and the unexpected always happens, how incapable must Man be of learning from experience!

GEORGE BERNARD SHAW

History repeats itself, and that's one of the things that's wrong with history.
CLARENCE DARROW

Pointers

*Gateway Charting
Lab:*

*Lab Unit 12:
Manual: Visualizing Data*

20.0

Overview

Chapter 7: Models, Visualization, and Pictures demonstrated that graphic or visual data can often be more useful or persuasive than written descriptions. Similar observations may be made about the results of graphic results and numeric models. Recall that the purpose of numeric models is to help users understand the relationships between complicated concepts. Unfortunately, even numeric models are often insufficient to provide truly clear understandings. A *graph* or *chart* can capture the essence of large or complex data sets.

The titles of this and several of the following chapters capture an important theme that runs through the remainder of this text: repetition. These chapters reach back to earlier chapters and present many familiar concepts in new contexts. There is no reason to

return to “square one” ! Simply reuse what you already know.

1. Although the word *graph* is often used to refer to the charts described in this chapter, I try to avoid that usage to avoid confusion with the computer science term *graph* defined in Section 6.5.3 and reappearing in Chapter 21. The term *graph* is used here when there is an especially strong tradition for such use, as in Cartesian graphs.

20.1

Visualization as Data Reduction

Simple collections of numeric data may contain much information, but tell the reader very little. For the information to be useful, it must be presented in a way that captures not just the raw data, but the essential details. For example, suppose you were thinking of purchasing a new computer. You looked at some machines, their prices, and their various features and are now attempting to compare two of them. For most people price will be a big factor. It happens that the two machines you like best are made by Orange Computers (for \$2,139) and Itty Bitty Machine (for \$2,611). Although it is not hard to tell which is more expensive, it may be much harder to grasp exactly how much more, or how much more in relation to the total cost. A figure such as Figure 20.1 can clarify the issue. A quick look shows that the Itty Bitty Machine is more expensive, but clearly not twice as much. The chart provides an easy-to-understand representation of the relationship between the prices.

People create charts for two reasons:

understanding, and

persuasion.

Actually the two reasons are closely related. Understanding is approximately the same as persuading yourself. Persuading often takes the form of “getting someone else to understand.”

People use numeric representations to provide a measure or means of comparison. Charts help people visualize that measure. People have at least three problems with understanding numeric relations:

comprehension of values,

visualization of relationships, and

human short-term memory limitations.

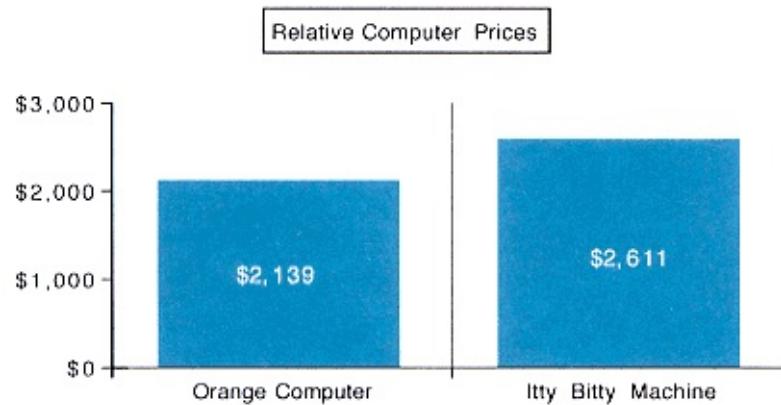


Figure 20.1
Using Size for a Simple Comparison

A reader of numeric data must first understand the values. When reading natural language text, most readers see entire words as single entities. They do not really see the individual characters. But the same reader sees numbers such as \$5.37 and \$13.01 as sequences of individual digits. The sequences must be translated into a number. The reader then must figure out the relationship between the values. You can't just glance at two numbers and say which is larger. The reader must first read each value carefully enough to understand the value represented. The algorithm for comparing is at least as complex as:

For each number

count the number of digits to the left of the decimal point.

If one has more

then that one is the larger number

otherwise compare the first digit . . .

This is true even when the exact values are not very important. To read numeric values you must read and comprehend each value before recognizing any relationships. *Chapter 10: Functions and Relations as Problem-Solving Tools* discussed approximation, a valuable problem-solving tool, but that doesn't seem to help with some comparisons. Even if you use approximate numbers, you must first understand the number before creating the approximation. The numeric values themselves do not show their interrelationships. Charts show these relationships quickly. When looking at two objects it is easy to see which is larger. The reader does not have to look at the individual values but can perform the comparison visually from the chart.

Finally, data collections often contain a great many individual values. For example, a list of the Dow Jones stock averages for the past year would contain about 250 values. Some days the average goes up and some days it goes down. Looking at a list of the daily averages for the entire year is not useful because no one can

remember that many numbers, especially while thinking about the relations between selected values. Analysis of the year would be very different if the daily averages leapt wildly from day to day, or if the maximum occurred near the beginning or the end of the year. The same data in a chart is much easier to understand.

20.2

Classes of Problems

Both numeric data and charts come in many forms. The appropriate tools and techniques for illustrating a given data set depend on the nature of that set.

20.2.1

Simple Comparisons

The relational operators (e.g., $<$, $>$, $=$; see Section 17.1.2) only allow for questions of the form: “is one larger than the other?” They cannot answer, “Is one much larger than the other?” To answer that question, either the writer or the

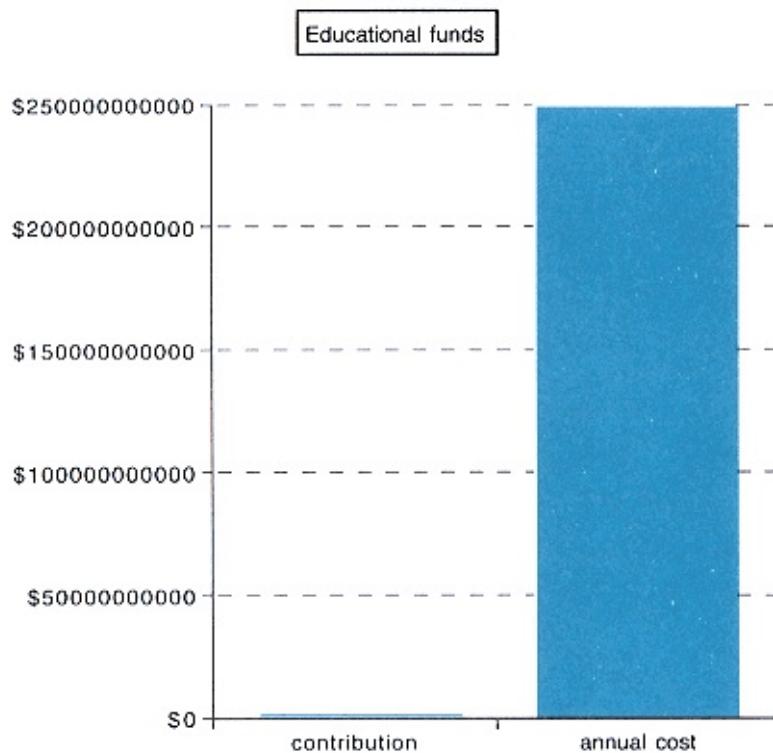


Figure 20.2
A Revealing Comparison

reader must perform the subtraction. Either way, the answer is still another number. Many people find problems of large numbers, such as the national debt, especially confusing. How often have you heard a discussion and after it is over found yourself asking, “Did she say ‘millions’ or ‘billions’?” Similar problems occur with any other quantity measured in units with which people are not very familiar. For example, a major benefactor, Walter Annenberg, recently announced that he would donate \$500 million to education. Wow! It turns out that the nation spends \$250 billion per year on public education. So, did Mr. Annenberg’s donation help the overall situation? Well, certainly he was very generous, and the donation will help some organizations immensely. The *bar chart* in Figure 20.2 shows the relative sizes of the two quantities. One quick look at the chart makes it clear that this large amount is really just the proverbial “drop in the bucket.” In general, any comparison between a small number of items in which the important questions

are of the form:

Which is larger?

Roughly how much larger?

What is the difference compared to the total size?

can be illustrated with a simple bar chart.

Comparing Several Values.

Sometimes, what we really want to know is “which (of several items) is largest?” or “Which are the smallest two values?” Bar charts work well for comparisons of several values. For example, Figure 20.3 contains a bar chart showing that more companies monitor employees’ computer work files than monitor their Email messages. On the other hand, Email messages do seem to be a major target for monitoring.² Notice that these are not mutually exclusive values; the total is well over one hundred percent.

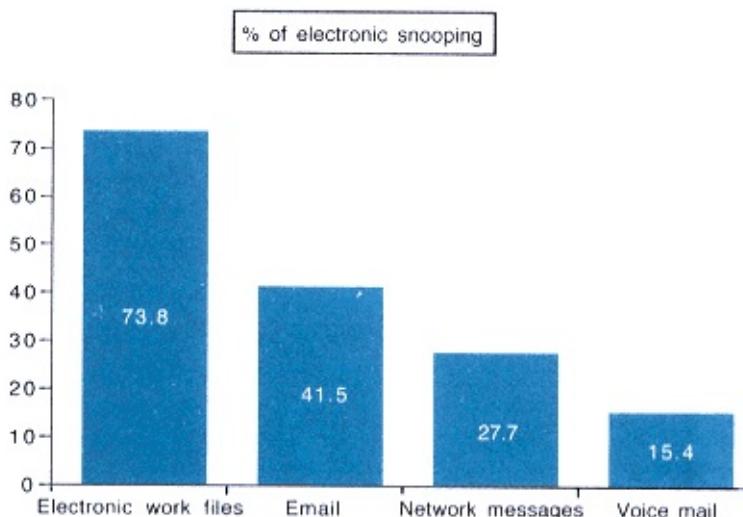


Figure 20.3
Of Companies That Monitor the Electronic Files of Employees,
What Percentage Monitor Each Type of File

Source: The statistics in this chart are from Piller, C. “Bosses with X-ray eyes.” *MacWorld*, July 1993, pp. 118-123.

2. Note that questions raised by these figures are now a major topic of debate in the computer and computer science communities. There is no

universally agreed on answer to the question, “How much privacy is an employee entitled to?”

20.2.2

Problems of the Whole

Many problems can be stated in terms of "the whole quantity." How is the cost divided up? Which group contributes the largest share? Is a given group's share significant? For example, the federal government receives income from many different sources and spends it in many different ways. Figure 20.4 shows the revenue from each source as portions of the total revenue. Figure 20.5 shows how it is spent again as portions of the total. From the first chart, it is easy to see that more than half of the total revenue comes from direct tax on individuals (income tax and social security tax), with borrowing representing the next largest source. Figure 20.5 shows clearly that the relative cost of actually running the government is an almost insignificant portion of the total even though it represents a very large sum (over a trillion dollars) in absolute terms.

These two examples are *pie charts*. Each shows "how the pie is divided up" or what each group's "piece of the pie" looks like. The essential feature of a pie chart is that it shows the relative proportions that make up a whole. The sum of the parts should be a relevant value. For example the sum of the two computer prices in Figure 20.1 is not a useful piece of information, nor is the relation of either price to that sum. Thus a pie chart would not be useful for simple comparisons. In effect pie charts translate a series of large numbers into fractions that total to 1.

The sum should represent a meaningful total against which the individual

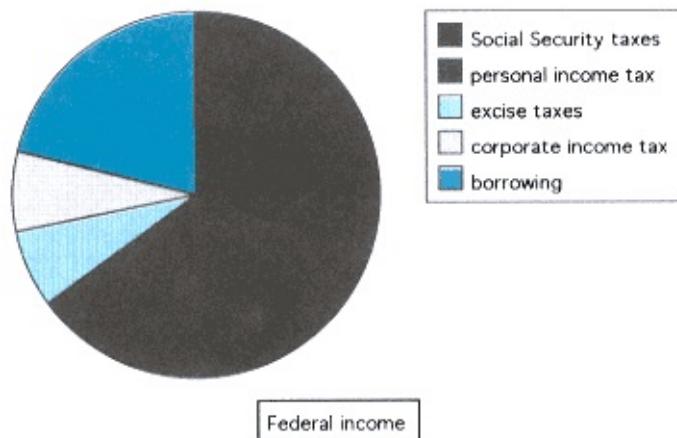


Figure 20.4

Where the Federal Government Gets Its Money

Source: The federal revenue and expenditure statistics are from *Form 1040* of the U.S. Internal Revenue Service, 1994.

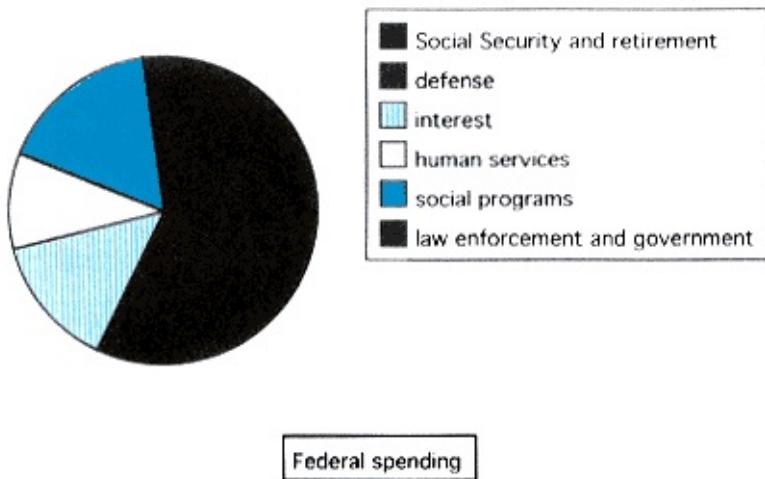


Figure 20.5

Where the Government Spends Its Money

Source: The federal revenue and expenditure statistics are from *Form 1040* of the U.S. Internal Revenue Service, 1994.

items are to be compared. Consider the funds received by a charity from its three largest contributors. Although the total contribution of the three may be an interesting number, the fraction of that number donated by each is not particularly interesting. A much more interesting comparison would compare the contributions to the total of all contributions, not just the top three. In general pie charts will very often need an “other” category for this purpose.

Finally, notice that pie charts are not particularly good for representing small differences such as 14% versus 16%.³ So unless the point of an illustration is that the quantities are approximately equal, avoid pie charts for such quantities.

Exercises

20.1 Create pie charts showing (a) the sources of your college funds, and (b) where you spend those funds.

20.2 For each of the following indicate whether a bar or pie chart would be better:

Bowling scores

Shoe sizes

(table continued on next page)

3. However, people are very good at recognizing halves, as well as near misses such as “just over a half.” Actually they are good at recognizing straight lines. Just over and just under a half appear as bent lines in a pie chart.

(table continued from previous page)

Costs of remodeling a kitchen

Cost of a beer at local bars

Distribution of funds at the end of a Monopoly game

The amount you spent on textbooks this semester.

20.3 Create bar charts for (a) the heights of your housemates and yourself, (b) the total area of each room in your house or apartment, (c) the ages of living presidents.

20.2.3

Trends

Sometimes the most important fact to be derived from a large set of values is the presence or absence of a trend. For example, Figure 20.6 shows a general trend over time. The chart shows the trend itself very clearly despite individual fluctuations in some years. When such small fluctuations are important, a simple alternative may capture the relevant aspects better. For example, most people hope that their income over a period of years will be steadily increasing each year (or at least most years) larger than the previous year, perhaps as in the chart in Figure 20.7. It is clear that the income has gone up each year, but it is also hard to tell how steadily. The *line chart* in Figure 20.8 shows the same information but in a form that makes it clear that the yearly change is not constant. Since the line connecting the annual income figures is not straight, income must not be increasing at a steady rate.

Notice also that the ordering of the columns in this figure is very significant: the order of the columns represents sequential years. The columns in both this chart and in Figure 20.3 are in order of changing magnitude, one decreasing and one increasing. However, the columns of Figure 20.3 have no inherent order; the chart would

be just as valid if “voice mail” had appeared in the middle. A reordered income chart would not be very useful for visualization because it would

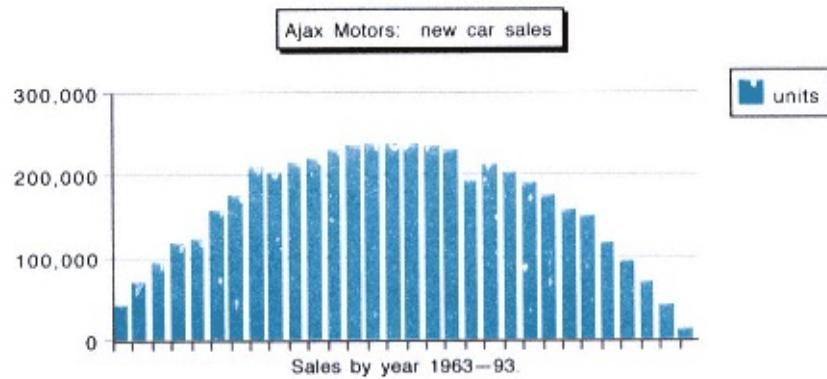


Figure 20.6
Use of Bar Chart to Show Trends

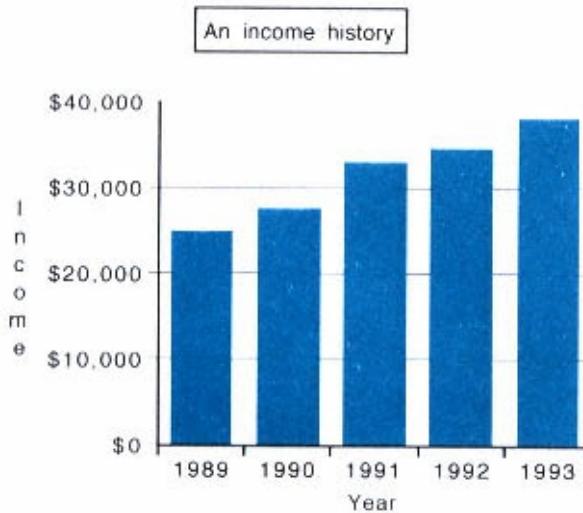


Figure 20.7
One Person's Income as a Bar Chart

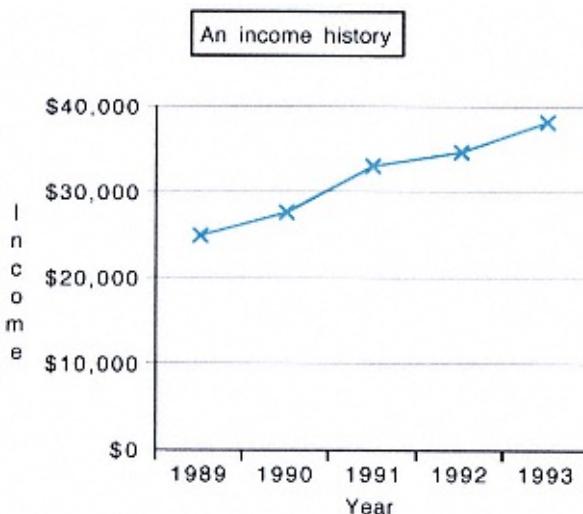


Figure 20.8
Income Growth over a Five-Year Period

not show the most important thing: the chronological trend. A *line chart* is especially useful for showing trends because the *slope* of the line explicitly shows the change from one entry to the next.⁴ Since the line is the focus of a line chart, you should never use a line chart for relations in which the order of the elements is not significant (e.g., the electronic surveillance data of Figure 20.3 or the revenue data of Figure 20.4).

20.2.4

A Quick Review of the Cartesian⁵ Coordinate System

Notice the relationship between the two charts in Figure 20.7 and Figure 20.8. The value points in the line in the second figure represent the tops of the columns in the first. The line merely connects those points, helping the reader visualize the trend. Graphically, the top of each column is a point that can be defined by two values: the column (year) and the height of that column (how much income). These are exactly the defining characteristics of a Cartesian coordinate system.

Although most readers of this text have seen and used the *Cartesian coordinate system*, for some, it may have been some time since they last used it. The Cartesian coordinate system is simply a way of representing relations graphically. Recall that a relation is a set of ordered pairs. Numeric relations can be thought of as points in two dimensions: horizontal and vertical. The horizontal position represents one field of the relation and the vertical component represents the other. Thus, the leftmost point on the line chart in Figure 20.8 represents the element: (year: 1989; income: \$25,000).

Traditionally, values to the right are larger and values that are higher are larger. The origin, or the point (0, 0) is in the center although regions of the chart containing no interesting points are often omitted. For example, the income chart contains no negative incomes. The origin is thus the left end (year 0) of the base line (0 income) of the chart. Each element of a relation is a point on

the chart such that the horizontal distance from the origin to the point is proportional to the size or magnitude of the one field of the relation, and the vertical distance from the origin to the point is proportional to the size or magnitude of the other field. Since 1989 is the first (lowest number) year in the relation it is leftmost (closest to the origin). The vertical measure of that point is also smallest, so it is the lowest point.

For functions, the independent values are traditionally represented by the horizontal-, or *X-axis* and the dependent variable by the vertical- or *Y-axis* of the graph. Even for relations that are not functions, the defining characteristic of the elements is usually represented horizontally. People tend think of “the income they made in a given year,” rather than “the year they made a given income.”

4. Students of calculus will recognize that this is essentially what the *derivative* of a function shows: how steep it is or how rapidly the function value is changing.
5. Named after the French philosopher, mathematician, and scientist, René Descartes (1596-1650).

20.2.5

Combinations of Values

Many problems are far too complex to distill the answer into a single chart of any type. Fortunately the basic graphing concepts can be combined to build more complex charts that convey the corresponding complexity.

Series of Sums.

Suppose you were charting your income over the past several years as in Figure 20.8. This time however, you had three different sources of income: your employment, your stock portfolio, and income as a freelance writer. In that case you might also be interested in the relative proportion of your income that came from each of the sources, and how that proportion changed over the years. Obviously no one chart of any of the types discussed so far can reveal all of the change in each category. For any given year, a pie chart can show the relative totals at any moment. A series of pie charts could show the changes, but such a series presents several problems. First, it requires multiple types of chart: a line chart for total income and several pie charts. Second, it requires multiple pie charts, but the eye is not particularly good at seeing and comparing the relative sizes of corresponding parts of separate pie charts. Finally, each pie chart seems to show the sum of the parts as the same pie. It would be possible, of course, to draw larger pies but that would not depict very well the changes from year to year in a given source of income (how do you compare a large piece of a small pie to a smaller piece of a larger pie?).

Consider the relationships and facts that a good chart should reveal:

1. Total income in a given year
2. Change in total income from one year to the next
3. Income from a given source in a given year

4. Proportion of income from any given source in a given year
5. Change in income from a given source from one year to the next
6. Trends in total income over several years
7. Trends in income from a given source over several years
8. Changes in the proportion of income from any given source over time

Items 1, 2, and 4 seem well suited for a line chart; item 6 seems well suited for a pie chart; items 7 and 8 seem to require a series of pie charts; and items 3 and 7 seem to need a set of line charts. The representation should capture the essence of the relationships between the data items.

Solution One:

Multiple Line Charts.

Several line charts togetherone for each source of income, and even one showing the total incomemcan show all of the important results. This captures the long-term trends for all absolute values, but does not capture the relative proportions. Figure 20.9 on page 354 shows these relationships. Notice that color makes it much easier to distinguish the individual lines. The chart represents all items except proportion and change in proportion pretty well. Notice in this case, that the disparity in magnitude between the

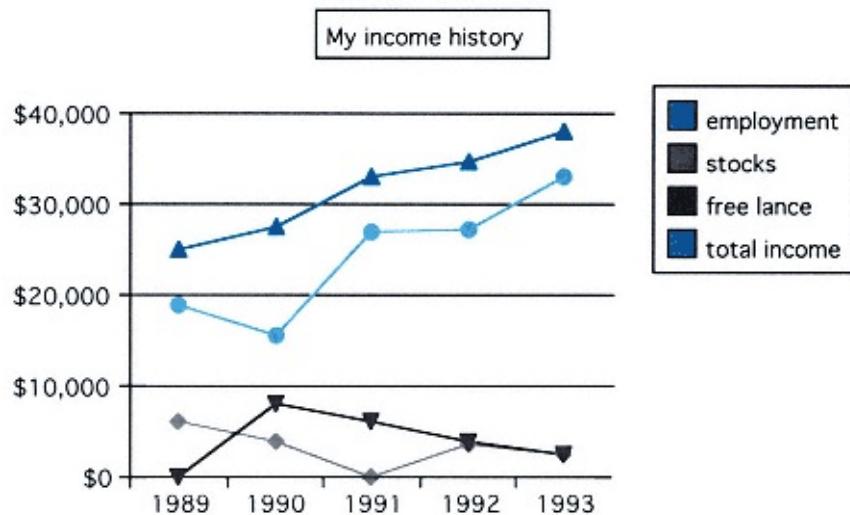


Figure 20.9
An Income History Showing Separate Categories

subcategories also interferes with the readability: the largest values must fit, but the smallest must still be visible.

Solution Two:
Composite Bar Charts.

The eye does not compare two pie charts very well, but it is good at comparing parallel lines or bars. Each bar of a single bar chart could be composed of the individual parts. Figure 20.10 shows such a composite bar chart. Note that it has only three categories since the total is implicit as the combination of the others. This chart facilitates comparison of both the proportion and absolute value of each category. Finally, if the most important aspect is the comparison changes in the relative incomes from year to year, a multiple bar chart such as that of Figure 20.11 is useful. Notice how the grouping makes some comparisons especially easy. Comparing the employment income is much easier than comparing the stock or freelance income.

20.2.6 *Functions*

Many charts show formal mathematical relations or functions.

Recall that a function is just a relation in which the value of one field uniquely determines the value of another. *Line charts* are ideally suited for demonstrating the relation between independent and dependent variables. For example, Figure 20.12 on page 356 depicts the values in Table 20.1: The line helps convey many of the characteristics commonly associated with functions, such as its slope or its continuity. A bar chart also works well for many functions, in which case you should always use one bar for each value of the independent variable, with the height (length) of the bar indicating the dependent variable.

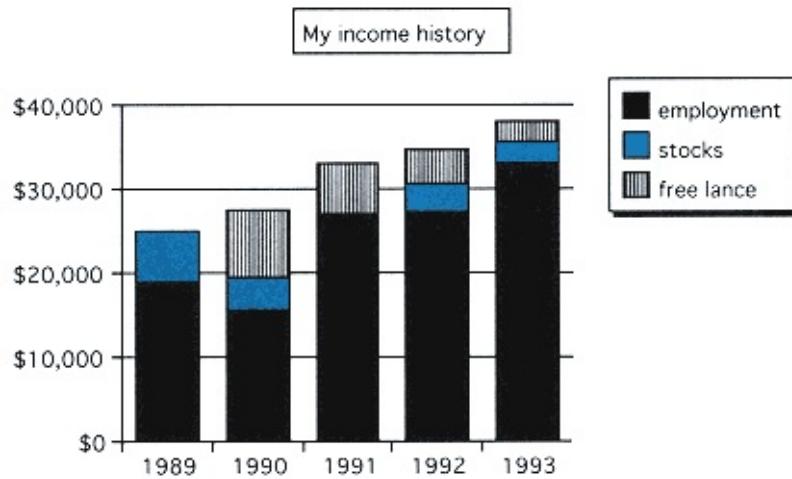


Figure 20.10
Income History as a Bar Chart

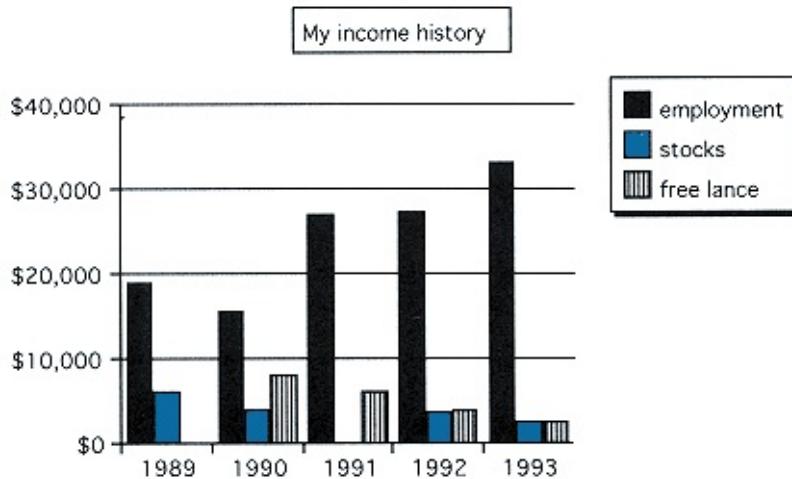


Figure 20.11
Multiple Bar Charts for the Income Problem

20.2.7

Irregularly Distributed Data

All of the examples thus far have assumed that the series is uniform in structure: the columns either represent distinct entities (e.g., Email and voice mail in Figure

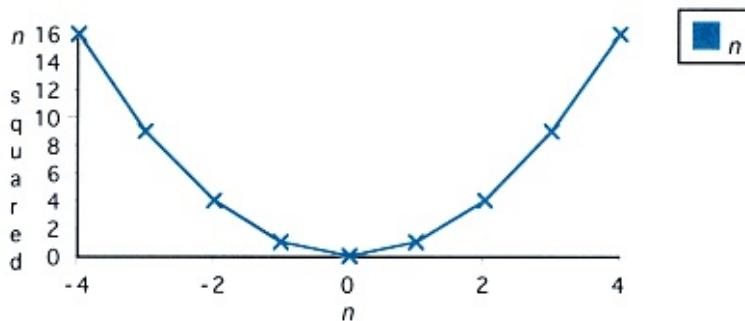


Figure 20.12
Graphing a Mathematical Function

20.3) or uniformly distributed numbers (e.g., calendar years in Figure 20.10). The independent or horizontal component either had no units or the difference between any two adjacent columns was uniform. Graphs are especially valuable when the sample data is not uniform. For example, suppose that an experimenter collected the values for Table 20.1, but for some reason couldn't use the same nicely distributed set of values for n . The resulting table might look like Table 20.2, with the corresponding chart in Figure 20.13. Notice that the data points (marked by X's) are not uniformly distributed horizontally in the chart. Rather, they are distributed according to the actual independent values. The lines of an *X-Y line graph* connect such nonuniformly distributed points. The resulting chart is almost as good as Figure 20.12, which was created from uniform data. Notice that the labeling is distinct from the actual data points. The implication for chart builders is that in a bar or line chart, the user provides labels (e.g., the years) and values (e.g., the income). For an *X-Y* graph the designer provides ($x-y$) pairs of points.

TABLE 20.1

n	n squared
4	16
3	9
2	4
1	1
0	0

1	1
2	4
3	9
4	16

TABLE 20.2

<i>n</i>	<i>n</i> squared
4	16
2.5	6.25
1.9	3.61
0.5	0.25
0	0
0.5	0.25
1.5	2.25
3.33	11.0889
4	16

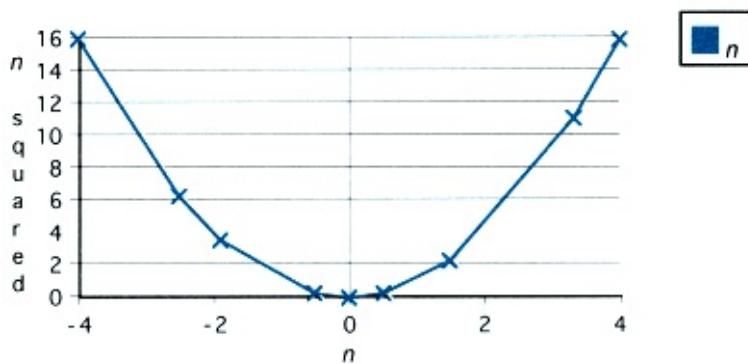


Figure 20.13
Function with Nonuniformly Distributed Data Points

In addition to variations in distribution, data may not be well ordered. In all previous examples, the data has been ordered sequentially by the independent variables. It is not always the case that data arrives so nicely ordered, as in Table 20.3 on page 358, which contains exactly the same values as Table 20.1, but in a different order. The user may sort the data, but this may not be desirable for various reasons. An *X-Y scatter chart* contains just the points corresponding to the *X-Y* values of the independent-dependent pairs without actually drawing the lines between them. Since the points are not connected by lines, the order doesn't matter as shown in Figure 20.14. The reader can usually visualize a line connecting adjacent points.

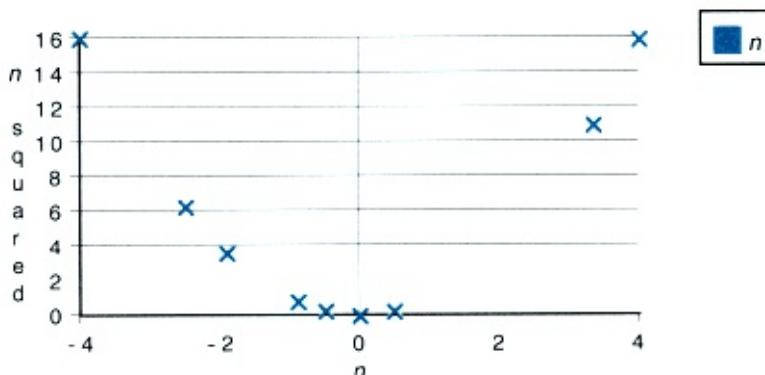


Figure 20.14
An *X-Y* Scatter Plot of the Function n^2

TABLE 20.3

n	n squared
4	16
4	16
0.5	0.25
2.5	6.25
0	0
0.9	0.81
1.9	3.61
0.5	0.25
3.33	11.0889

$X-Y$ line and $X-Y$ scatter plots can be far less misleading than simple line or bar charts when the independent data is not uniformly distributed. For example, if the data in Figure 20.11 were for the years 1980, '81, '83, '92 and '93, the bar chart would miss one important fact: a long period of small increases. Since the total increase for the nine years from 1983 (third column) to 1992 (fourth column) was about the same as the increase in other years, the average increase for those years must have been much smaller than for the years before or after. The graphic representation should capture the variations in both dimensions. In this case, proper representation requires both income and time. Figure 20.15 compares a simple line graph and an $X-Y$ line plot of the total income for these years.

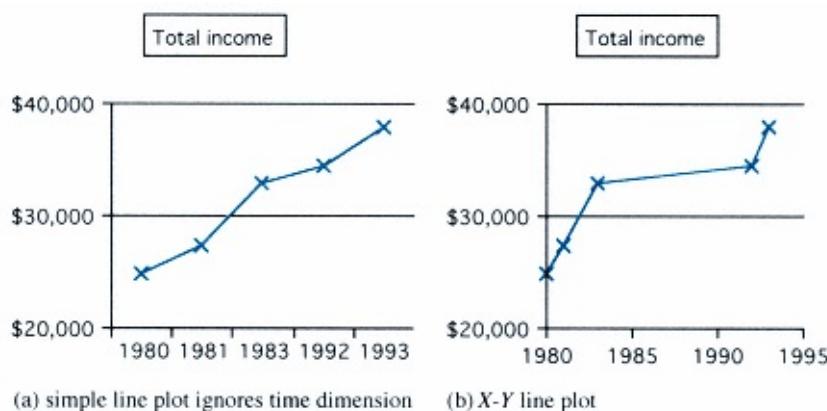


Figure 20.15
X-Y Graphs Can Be More Revealing

20.3 Misleading Charts

In general, visualization attempts to reduce data to fewer but more meaningful items. It is also a form of approximation. Section 10.1.1 described approximation as a very good tool for understanding. Graphing is actually just such an approximation or abstraction. The size of a bar shows the approximate value. Two bars are an abstraction of the concept of relative size. Charts should help visualize data, not obscure it. Yet obscurity is just what some charts accomplish.

20.3.1 *Misleading Proportion*

Figure 20.16 is produced by the town council of Gregsville, showing recent changes in government spending by the town. The chart seems to show that government spending is falling at an admirable rate (which is, after all, exactly what the town fathers were attempting to show). Unfortunately, the spending is actually decreasing at only 0.1% per year, falling only \$3000 in three years. Charts such as the one in Figure 20.16 have earned the nickname "politician's chart" since they tend to do more to deceive than to help the reader understand. (Can you find another example of the politician's chart earlier in this chapter?) Figure 20.17 shows what happens when the chart is shown more honestly. At first glance, it may seem like the first version was better because it actually showed the decrease, while the latter version missed the decrease entirely. But that is precisely the point: the decrease was so small as to be insignificant a fact that the second chart shows and the first one missed.

Charts are supposed to elucidate the relationships between values. That means that the magnitudes in the chart (bar size, line height, etc.) should reflect the

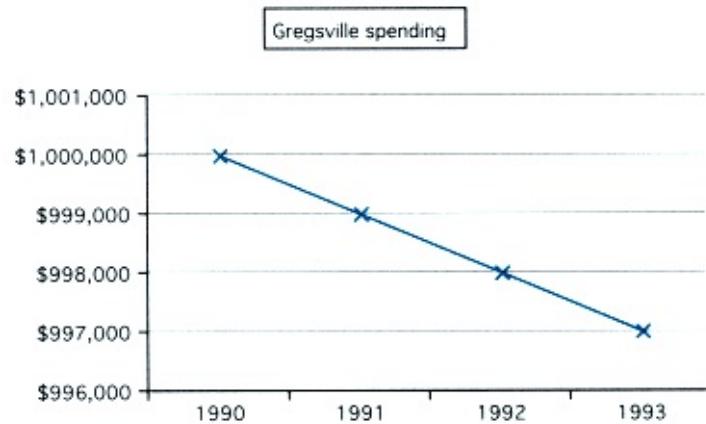


Figure 20.16
A Politician's Chart

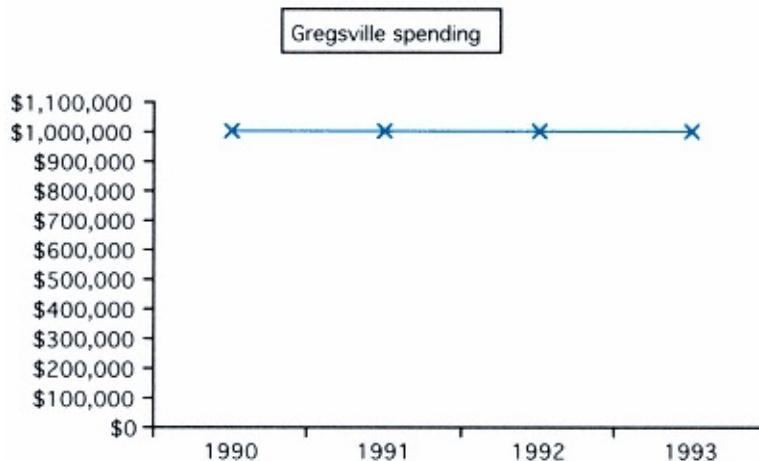


Figure 20.17
A More Honest Version of Figure 20.16

relationships that exist in the original data. The scale for a bar or line should normally start at zero. If it is really impossible because of the size required, be sure to make it clear that you have shortened the chart.

20.3.2

Rules of Thumb for Using Visualization Tools

Although there are no absolute “do’s and don’ts” for creating charts, several rules of thumb will lead to better, more easily understood charts:

Human eyes are good at recognizing parallel lines. This implies that they can see the relations between the lines in a multiple-line chart. They can also pick out the tallest line relative to a base point.

Color is a valuable tool for distinguishing regions. It is much preferable to changing marker shapes alone. (But also keep in mind that a significant minority of people are color blind. Use of color and shape together may be helpful.)

Objects are easier to distinguish if they are close together. But they are easier to recognize if they are actually separate.

Avoid extra lines, details, and patterns that do not add directly to

the understanding. In a word: simplify.

Be wary of measures with unclear units. For example, computer magazines often compare the features of new computers. They sometimes include a chart showing the relative speed of two machines for performing a given operation (e.g., speed for scrolling through a document). Without careful labeling it may be very difficult to interpret the chart. Does a longer bar mean that the machine scored high on a speed test (i.e., it is fast), or that the machine took a large amount of time to complete the task (i.e., it is slow)?

Exercises

- 20.4 Build charts for the functions created in Exercises 15.10, 15.12, 16.3, and 16.10.
- 20.5 Build the graphical version of Exercise 16.13.
- 20.6 Build the chart that illustrates integration via the method of Exercise 16.22.
- 20.7 From your newspaper get the Dow Jones Industrial averages for the past week. Graph these figures.
- 20.8 Repeat Exercise 20.7 but for the Nasdaq and Standard and Poor's averages in a single chart. Comment on any problems encountered.

20.4

Charts as Tools for Understanding Mathematical Relations

20.4.1

Exponential and Logarithmic Values

Consider the chart in Figure 20.18, which represents values collected in an experiment. Most scientists will recognize the shape quickly as *exponential* (y is proportional to 2^x or e^x)⁶ or approximately so. But actually it is not clear whether it is exactly exponential, or perhaps a little above or a little below. One way to visualize how close the function actually is to exponential would be to superimpose the graph of

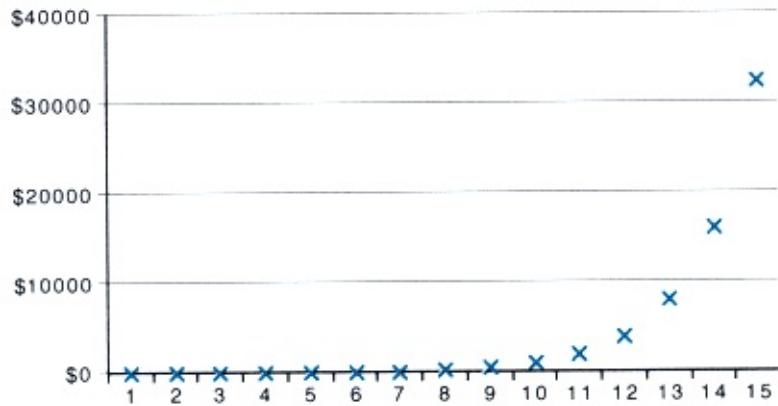


Figure 20.18
A “Close to” Exponential Curve

6. Most scientists use the base e for comparing exponential series of numbers. Computer scientists tend to use the base 2. The results have exactly the same shape.

$$y = e^x$$

on Figure 20.18 as in Figure 20.19. That figure has two problems. You would need to generate the second graph, and that might obscure the data. In the example, the second graph was so much larger that the original was barely perceptible. An alternative method is to graph the results logarithmically. Since

$$\log(e^x) = x,$$

the result of plotting the log of the y -value against x should be a straight line if the function is exactly exponential. Since the graph in Figure 20.20 curves slightly downward,⁷ it is clear that the function is slightly less than exponential. The difference is really dramatic at the left, but the important values are the ones at the right. Notice that even though the function is only slightly less than exponential, that fact is easy to see in this graph.

20.4.2

Graphing and the Calculus

Graphs or charts can make an excellent tool for checking your understanding of the calculus. Figure 20.21 shows the function:

$$f(x) = -x^2 + 10$$

and its derivative

$$f'(x) = -2x/dx$$

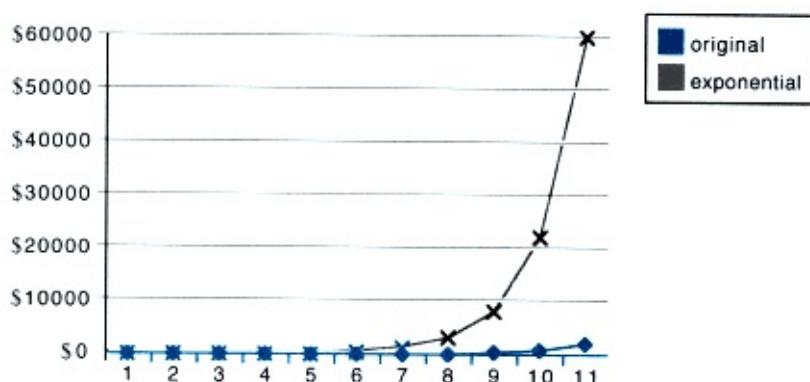


Figure 20.19
Comparing a Function to a Known Baseline

7. That is, the function has a negative second derivative.

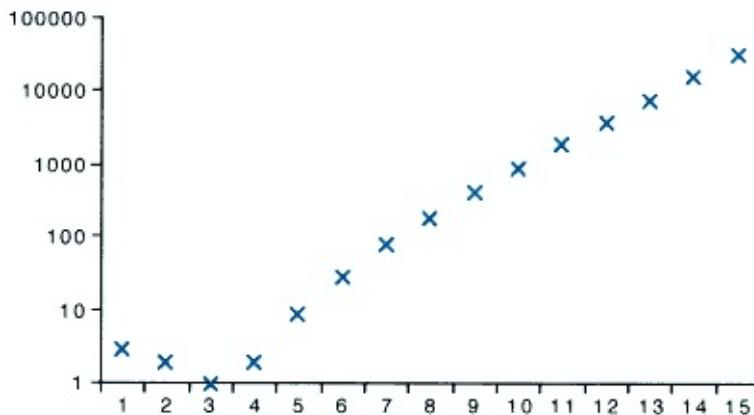


Figure 20.20
Logarithmic Graph of the Function Shown in Figure 20.18

The graph clearly shows that the function reaches a maximum at exactly the point where the derivative becomes zero. Few spreadsheets include a derivative function, but you can use them to check your results visually: graph a function and the proposed derivative and see if the maxima and minima correspond to zero-crossings. In addition, easy-to-use mathematical packages, such as *Maple* or *Mathematica* now provide automatic integration and differentiation.

A similar technique can illustrate the definition of integral as the “area under the curve” or the sum of the D-x’s.

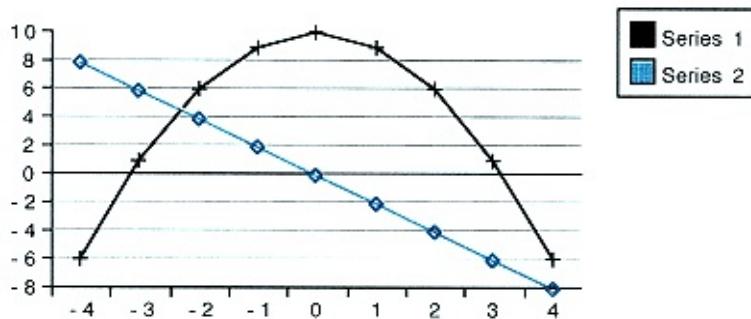


Figure 20.21
A Function and Its Derivative

Exercises

20.9 Plot the function $f(x) = x^3 - 2x^2 + 3$ and its derivative.

20.10 Graphically compare the functions ex and $2x$.

20.5

Summary

Graphs or charts provide excellent aids for making comparisons or visualizing the mathematical relationships. However, it is essential that you select the appropriate charting tool for a given task. Otherwise you may create a misleading chart.

Important Ideas

chart graph pie
 chart

bar chart line chart xy
 graph

trend series function

approximate coordinate
 system

21

Graphs and Hypermedia

O, what a tangled web we weave, When first we practise to deceive!

SIR WALTER SCOTT (*Marmion* (1808))

Pointers

Alternate chapter: Chapter 22: Telecomputing1

Gateway Labs: *Hypermedia navigation Gopher*

Lab Manual: Unit 13: Graphs, Hypermedia, and the Internet

21.0

Overview

Data may have any of several natural organizations: linear, matrix, hierarchy, and so on. Some data sets have more complex organizations. A given data collection may even have more than one natural organization. Previous chapters discussed organizational tools ranging from word processors for *ad hoc* organizations to spreadsheets for matrix organizations. Each tool forced a single structure on the document. *Hypermedia* documents may simultaneously have more than one organization. Users may view data in any of several dimensions and easily switch from one viewpoint to another. Most readers of this text have already seen and used many hypermedia documents: both the online help provided by many applications and the *Gateway Labs* accompanying this text are hypermedia documents.

21.1

Graphs

A *graph* is a diagram, like a tree: it has nodes connected by *edges*. But a graph does not have the restrictions dictated by the definition of a tree. Each node may

1. The material in this chapter and *Chapter 22: Telecomputing* are quite intertwined. Local situations, such as availability of software, will dictate the best order for reading this material. The two chapters can be read in either order. In fact, these two chapters would be excellent candidates for hypermedia.

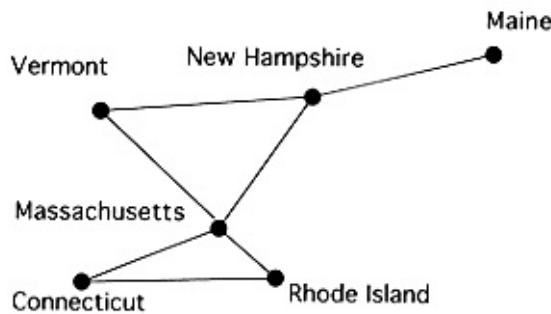


Figure 21.1
A Graph of the New England States

be connected to any other node, not just to its parents and children. In fact, since a graph is relatively unrestricted, many tree-concepts and terms such as parent, child, and branch do not easily apply. Figure 21.1 contains a graph in which each node represents a state in New England. An edge between two nodes indicates that the two states share a common border. A road map is also a graph: the nodes represent cities, the edges connecting them, roads. Graphs can represent an arbitrary organization, not just a linear or hierarchical organization. Section 11.2.2 represented the algorithms corresponding to arithmetic expressions as trees. Graphs can depict a broader set of algorithms: each node represents a step and each edge the chronological relation between two steps.

21.1.1 *Embedded Organizations*

Edges in a graph often represent more than one sort of relationship. If the nodes in Figure 21.2 are the members of a family, the solid lines represent the usual parent-child relations; the dashed lines represent sibling, cousin and aunt-niece relations. Notice that restricting the graph to just the parent-child links produces a simple tree. We say the tree is an *embedded structure* within the graph. The graph can also be thought of as representing multiple views of information: one view

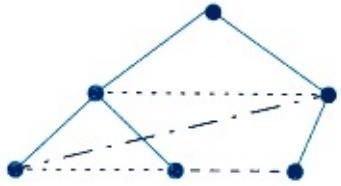


Figure 21.2
A Graph of Family Relations

corresponding to each embedded structure. Thus, the graph in the figure also seems to show a generational or sibling-oriented view of the relationships. Every graph contains at least one tree structure (called a *spanning tree*) that includes all of the nodes but not necessarily all of the edges in the graph. It is not always possible to find a linear list that includes each node (with no repeated nodes).

21.1.2

Limitations of Graphs

The universality of a graph can also be a drawback. Unlike hierarchical and linear organizations, graphs do not necessarily have an obvious structure. The lack of root and leaves means there is no obvious search strategy. The user must select one path from many on the map. Until relatively recently there have not been any computerized tools for manipulating or following arbitrary graphs. Tools that provide such capabilities are collectively called *hypermedia applications*.

Exercises

21.1 Why aren't the terms *root*, *leaf*, and *branch* meaningful in a graph?

21.2 Not all algorithms can be represented as trees. Some require more general graphs. Under what situations does an algorithm require a general graph rather than a tree?

21.3 Draw a graph depicting the rooms of your house. Create edges between each pair of adjacent rooms.

21.2

Related Tools

Physical representations necessarily impose a primary ordering on data and information. The physical organization of a book even a

book with a hierarchical organization such as this one is essentially linear, from front to rear. Each chapter logically and physically follows the previous one. Sections within the chapter also follow in a sequence. Unfortunately, not all users of a book want the same order. Even a given user does not want the same order on all occasions. One order may be best for investigating a new subject (e.g., the usual use of a textbook), but a very different order may be best for looking up the details of an item (e.g., using a text as a reference manual). *Hypermedia* allows a user to view a single document in multiple or more complex organizations. Although no single tool discussed previously supports all aspects of hypermedia, several familiar tools embody essential characteristics or ideas.

21.2.1

Multiply Ordered Databases

Database applications (see *Chapter 8: Relations, Tables, and Databases*) allow a document to be viewed in multiple orders. The application allows dynamic

reordering of the displayed material. A user can request new organizations for the data at any time. For example, student grade information may be organized alphabetically by student for mailing grades, alphabetically/numerically by class for creating class lists for instructors, in decreasing numeric order for finding the valedictorian, and so on. The database application serves as a tool for altering the view of the data currently available to the user.

Limitations of Databases.

Sorted data is, by definition, in order. Resorting creates a new organization based on a different ordering scheme. Database systems do provide flexibility through sorting, but sorting, as a provider of multiple views, has at least three limitations:

1. Sorting a large database requires considerable time perhaps more time than a user is willing to wait. Viewing data in two separate organizations may require frequent switching between those orders. Each change of viewpoint requires re-sorting. The alternative maintaining two copies requires extra storage space and increases the probability of error.
2. Sorting requires a quantifiable measure that applies to an entire database (say, size, age, or alphabetical order). Unfortunately, there may be no quantifiable measure corresponding to the order the user wants. For example, the user may want frequently accessed items to be stored first. Or she may want to access descriptions of parts of a picture. These concepts do not provide a basis for sorting.
3. Within a single search the user may want more than one organization. For example, a travel agent may want to search for a particular city, and then within that city to look for the cheapest car rental.

21.2.2

Indices and Tables of Pointers

Traditional (paper) books provide alternative views of document order through the use of indices and tables of contents (see Figure 21.3). Although a document itself is linear, a table of contents views the book as a hierarchical structure. An index views the book as an alphabetical list of topics. Each entry points to all of the sections on a given small topic. Both tools are lists of pointers much like the pointers used throughout this book; both serve as tools for selecting the current view of the document. Both help a reader get to a particular page in the book. Using an index or table of contents, a user can find virtually any desired point in a book. Indices and tables of contents do differ in some essential ways. The table of contents lists chapters in the same order as the book, but the order of entries in an index is independent of the book order. The table of contents contains exactly one pointer reference per entry; the index can contain any number of entries, even references to other lines of the index.

Limitations of Indices and Tables of Pointers.

Unfortunately, static indexing tools are not as powerful as many readers would like. A pointer tells the reader where an entry is located but does not help the reader get to that location.

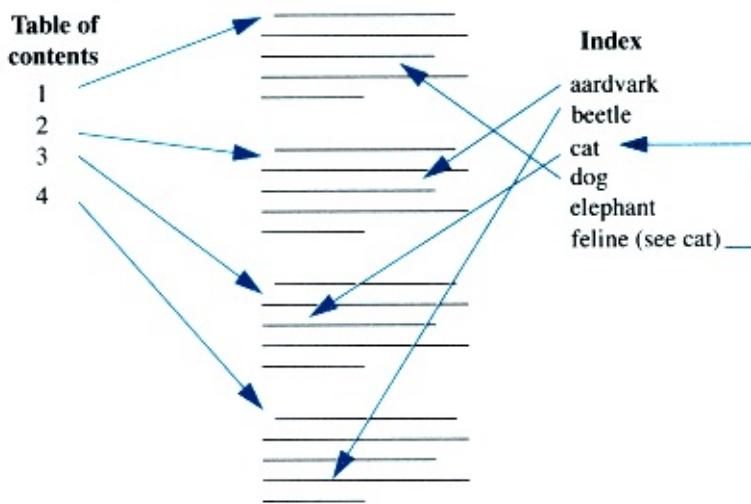


Figure 21.3
Two Forms of Index Structures

This sounds petty, but it can be significant when dealing with large or complex searches. Suppose you are reading a book about King Arthur and the knights of the Round Table. You find a reference to Tristram. Possibly, the name may seem familiar, but exactly who he was may seem elusive. If the book has an index, you can go to the index and look him up. An entry in an index may contain several references, but few indexes indicate the potential value of each entry. The user may be interested in Tristram: the lover of Isoud, the nephew of King Mark, the master of Dinadan, or the knight of the Round Table. The index does not say which entry tells about which aspect of his life. Consider the process of searching for Tristram's identity.

Mark your current page location. (You will want to return eventually.)

Find the index.

Find the entry for Tristram.

Read the list of pages where Tristram appears.

For each entry for Tristram in the index

(or until you satisfactorily identify him):

Mark your place in the index.

Go to the appropriate page in the text.

Scan the page for the relevant material.

Read the relevant section.

Return to the index.

Return to the page you were at before you were distracted.

The task is very user-intensive. The reader needs to move between sections located throughout the book, but at any point she is looking at a single section. The reader

could use some help in getting to the desired page, returning to the original location, and keeping track of where she has been.

21.2.3

Voice Mail

Voice mail systems have recently replaced many simple telephone answering machines. These systems allow the caller to navigate through departments within an institution and the call-recipient to navigate through her collected messages. She selects activitieslisten to any of several received messages, delete received messages, or record new answer messagesby pressing buttons on the telephone handset. Depending on the user's input, the answering system provides new sets of options. Many systems allow the user flexibility to request that the list be repeated, select an option before they are instructed, or other options. The voice mail system should be viewed as a tool that helps the user navigate through a collection of possibilities.

Limitations of Voice Mail.

Many people (especially among callers to voice mail systems) find such systems annoying. Fortunately, the worst problem is not an inherent problem of hypermedia, but is the result of poorly designed systems. For example, some systems do not allow any form of Undo or backtracking. Instead, the caller must hang up, redial, and restart the process. Limitations of human short-term memory cause additional problems: if the system presents the user with too many choices, the user will likely forget the first item on the list before hearing the last. Finally, notice that sound is intrinsically linear, forcing some degree of linearity on a system that is attempting to offer hypermedia capabilities. The user must often wait for instructions.

21.3

Dynamically Linked Documents

A *hypermedia* document is any document that allows the user to impose arbitrarily complex and concurrent multiple organizations on data. Each of the above examples provides a start in that direction, but each provides only part of the solution. A hypermedia system provides the graph-like capabilities found in each of the above systems: the user can view data from different perspectives, move through data in any order, automatically move from index to referenced page and back, or change tasks at any time.

The prefix *hyper* means "above or beyond." A *hypercube* has four or more dimensions certainly beyond the normal number; a hyperactive child is more active than reasonable, and a person with hypertension has high blood pressure. *Hypermedia* has more dimensions or structures than other forms of data. A hypermedia user can view a document in many dimensions, and even change viewpoint part way through. Section 5.5 describes written material as one-dimensional (overlaid onto two-dimensional paper). Spreadsheets and databases are inherently two-dimensional because the user can traverse the material either by

row or by column. Paper media are inherently limited to two dimensions. Most attempts at multiple dimensions actually superimpose one representation on another, either in space (e.g., an outline) or time (a database). Hypermedia enables truly multiply dimensioned documents.

21.3.1

Dynamic Links

The principal feature of any hypermedia document is the *dynamic link* or pointer. When the user selects a dynamic link, the current context immediately changes to a new location specified by the pointer. Imagine a table of contents, with dynamic capabilities. When the user selects Chapter 21, the book immediately opens to page 365. The first page of each *Gateway Lab* is just such a table of contents. With a single click, the user “moves” to a given section or the lab.

21.3.2

Interaction

The buzz word for all hypermedia documents is *interactive*. A hypermedia document must respond to user requests. This means that all hypermedia documents are, in effect, living documents (See Section 8.3). The document may not actually change as do database or spreadsheet documents, but it must provide responses to user requests. The user must be able to navigate between sections or perspectives by selecting pointers.

21.3.3

Visualizing Hypermedia

Probably the worst way to learn about hypermedia is to read about it in a noninteractive textbook. For this reason many hypermedia tools provide excellent introductions and tutorials. The way to learn about hypermedia is to explore it. And hypermedia provides the perfect presentation tool.

Browsing.

Hypermedia uses the metaphor of *browsing* through a document or book. In the language of hypermedia, the reader *goes* to a new location, as you would “go to” another page in a book. Technically we should probably say “a new page is displayed.” But it is easier to talk as if the user (or the application) actually changed locations. At any moment, the user is at a location (e.g., looking at a page). In the default model a reader can look ahead to the next page or back to reread the previous page.

21.3.4

Familiar Examples of Hyper-Documents

Hypermedia applications come in many forms, with varying degrees of support for dynamic linkage. Although the term may seem new, you have already seen several examples of hypermedia.

Help Files and Tutorial Programs.

Many applications provide help in the form of a hypermedia document (this is certainly true for both *Windows* and

Macintosh operating systems). Usually the user can search for help via both a topic list (table of contents) and a keyword index. Figure 2.1 illustrated one such interactive hyper-help document. Even applications that provide no other hypertext capabilities often have hypertext support files. Usually, the user can even jump to instructions on how to use the help facility.

Augmented Word Processors.

Some word processors can create hypermedia versions of simple text documents. This feature helps authors of large documents (e.g., a book) locate and cross reference the final document. Of course once a document is transcribed to paper, there is no way to take advantage of the hypermedia properties.

Games.

Computer adventure games of the *Dungeons and Dragons* genre are also complex hypermedia documents. When the game player selects a door, the game delivers the user to a new room or scene. Each scene is a node and each transition represents a dynamic link.

21.3.5

Common Pointers

From the user's perspective a hyperlink is simply a mark or *button* in the document. Selecting the link takes the user to the indicated place. The link may be a traditional computer button, a highlighted section of text, or even a picture or section of a picture. The links in Figure 21.4 are identifiable by emphasis (e.g., underlined or bold), or as the picture icons.

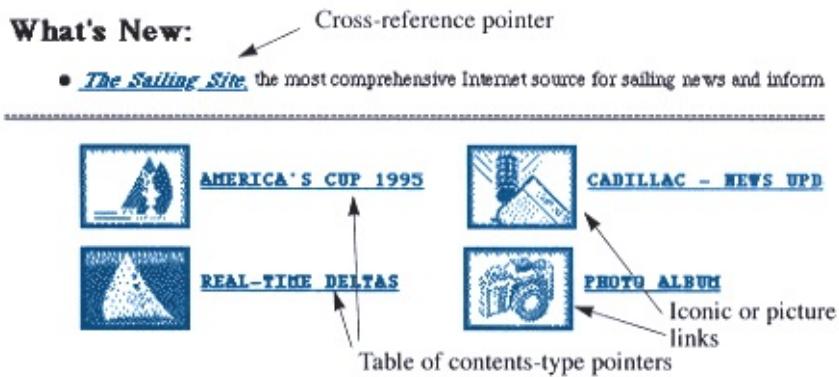


Figure 21.4

A Hyper-Document Page Containing Typical Pointers

Source: This segment is from *The America's Cup* Home Page, and is copyrighted by Science Applications International Corp.

Exercises

21.4 Find a hypermedia help file for your computer system.

21.5 Check the help tools of the applications you have used in this course. Do they appear to be hypermedia tools?

21.6 How could you determine if your word processor supports hypermedia? If it does, find out how it works.

21.7 Find a computer game. Make a list of the hypermedia links within the game.

21.8 Draw a tree structure representing part of your file directory.

Now add a few aliases to your directory and convert the tree to a general graph by including the corresponding links.

21.4

Navigation in Hypermedia

Most hypermedia documents can be read sequentially from first word to last, just like any other document. Dynamic links make other orderings possible. They also place the burden of navigation on the user. Fortunately “burden” is too disparaging a description. Navigation is very straightforward. Several options are almost always available.

21.4.1

Linear Structure

The user may view a document as a traditional linear document, reading the document from front to back. To make this simpler, many documents are divided into segments, called *pages* or *cards*. The names suggest that a segment might hold one page of a book. Logically, a hypermedia page is more like a chapter: one logical segment. The term *card* suggests note cards such as the ones a public speaker might use to keep all the notes for a speech. The first

page is called a *home page* or just *home*. The reader can look at the entire document starting at the home page and moving sequentially through the pages. At each page, she can move either after reading the entire page, or after deciding not to read it for some reason (e.g., the page is too difficult or not relevant).

Forward.

Forward links are much like turning a page or skipping ahead to the next section. Usually, a forward step moves one visual or logical unit, such as one page, one chapter, or one screenful of material. For example, if this were a hypertext document, a forward arrow placed right here might move the reader to Section 21.4.2.

Traditionally, the symbol for a forward move is a right-pointing arrow (P). Sometimes single and double arrows are used to distinguish small steps (e.g., next page) and larger steps (e.g., next chapter).

Backward.

Backward movements are a sort of Undo of forward steps: “Move back to the last section. I want to reread it.” Although not invoked nearly as often

as the forward move, this capability is essential even for a one-dimensional text document. The usual symbol for a backward link is a left-pointing arrow (Ü).

Forward and backward links by themselves enable a user to read an entire document, but used alone, they lack the flexibility characteristic of hypermedia: the reader may only read the document through from front to back, much like scrolling through text.

21.4.2

Hierarchical Structure

The table of contents model suggests another strategy for navigation: the *hierarchical search* (see Section 6.4.2). Treat the document as a tree. Start from a general root and search through increasingly specific pages. The home page usually contains a table of contents. Selecting an item in the table of contents moves the user to that section. Often the first page of each section contains pointers to all of its subsections.

[Return to Table of Contents.](#)

The user may want to return to the table of contents, or more generally move up a level in the hierarchy (i.e., move to the

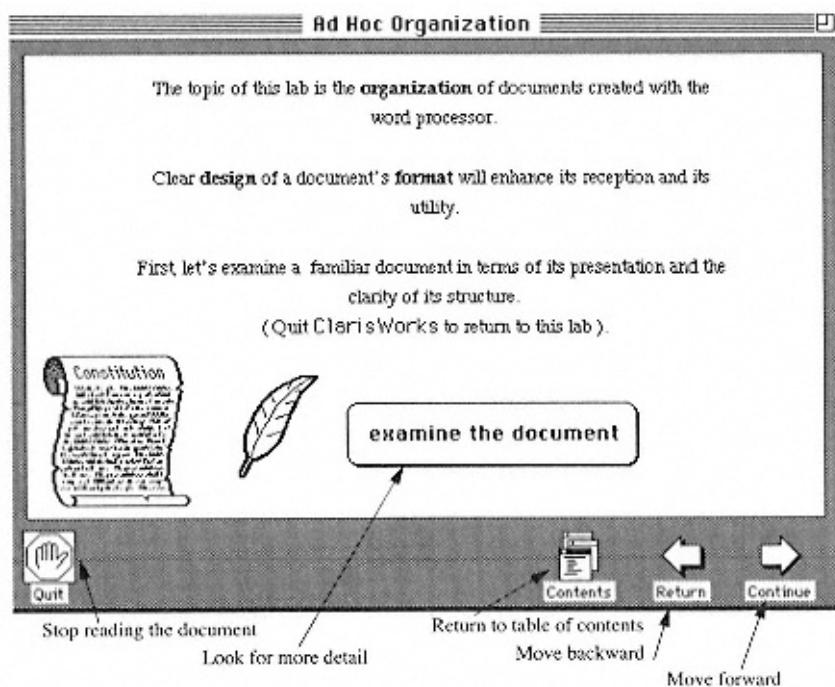


Figure 21.5
Links in a Familiar Hypermedia Document

parent node). For example, if this text were a hyper-document, a user at the current location (Section 21.4.2) could move up the section (Section 21.4), and chapter (Chapter 21). An up-arrow (↑) or a bent arrow usually means “move up a logical level.”

21.4.3

Generalized Pointers

Graphs need not be restricted to the more traditional data structures. Some common forms of pointer include:

Cross Reference.

Many keywords within a document are themselves pointers to sections of the document describing the names section. Cross references logically equivalent to parenthetical comments such as “(see Chapter 21)” are often denoted with emphasized (e.g., bold) text.

Help.

Help-links take the user to extra material that is not in the main stream of the text flow. The user only sees this material if it is needed. Help is much like a reference in a text of the form “for more specific information see . . .”

Pictorial Information.

Not all information is best represented as text. Pages of a hypermedia document may also contain pictures. Consider the task of an architect or real estate salesperson who wants to describe a new house. A sales pamphlet could contain a picture of each individual room, even pictures of each room viewed from each angle. How should the pictures be ordered? Largest room first? Prettiest room first? Perhaps it should be organized in the order clients would view the rooms while walking through the house. But of course each client may prefer to view the house in a different order. A hyper-document can allow exactly that. The document

might have links from each room moving the viewer to the adjoining room.

21.5

Hypermedia Applications

Several forms of application now use hypermedia documents as their principal data type. They have widely varied purposes but share a common underlying structure.

21.5.1

The World Wide Web

The *Internet*, a world-wide network connecting computers at virtually every university, government department, and large corporation in the world, is described in Chapter 22. Amazingly, hypermedia tools enable even neophyte users to access vast quantities of information from Internet sites around the world.

Hypermedia applications make jumping to documents on the other side of the world as easy as jumping to other points within a single document.

For the casual user, the *World Wide Web* is the single most important Internet tool available. Indeed, since “the Web” can provide a useful interface to most other net applications, it may be the only net tool many users will ever need. Essentially, the *World Wide Web* is a very loosely interrelated set of *hypermedia* documents. Pointers may point not only to other pages within the single document, but to pages in other documents. More importantly, these other documents can be located on other machines anywhere in the world. The user need not know the location of these documents, or even if they are in the same document or another. Instead the user follows pointers from one document to another, using a tool called a *web browser*.²

Web Data.

The information on the *World Wide Web* is as diverse as it is voluminous. Users can easily find data about computers and computer applications (e.g., prices or answers to common problems), universities (e.g., admission guidelines), television shows (synopses of soap operas), recreational activities, travel destinations, or chocolate. Web pages include computerized versions of magazines and newspapers, contest entry blanks, and recordings by country western singers. Links on the Web can take users to data in the form of:

directories

documents

specific points within documents

pictures or sound clips

specialized services

Most of these are straightforward: the user selects a keyword or button and the browser takes her to the new document. The only restriction is that the machine must be connected to the Internet

(most machines in college and university labs now have such access).

Web browsers also provide users with an easy-to-use interface for the more specialized services described in the next chapter. For example, a user may ask the browser to send Email to the author of a document, or subscribe to a magazine.

21.5.2

Gopher

Gopher is also a network-wide hypermedia service. It is older and more established than the *World Wide Web*, but is somewhat more limited. In particular, *Gopher* links are always document pointers. That is, each link takes the user to a new document or folder, never to a new location within the existing document.

Gopher lacks the flexibility of most Web browsers: pointers usually appear as documents or folders, never as parts of a picture. On the other hand, *Gopher*'s document orientation helps the user take advantage of the hierarchical structure

2. Popular Web browsers include *NetScape* and *Mosaic*.

used for file directories and folders. This serves the user in two ways: Searching for data resembles looking though an outline. At each level the user focuses on a narrower topic. Since some file organizations are reasonably standard, the user can often tell where to look for information.

Like Web browsers, *Gopher* allows users to access many of the more specialized services described in the next chapter.

21.5.3

Hypermedia Programming Environments

Programming is not restricted to either the spreadsheet model of recent chapters or the so called higher-level languages described in Section 24.0. Some applications³ enable designers to create their own hypermedia documents that include arbitrarily complex responses to user input. For example, a tutorial program might ask the user to solve problems, and select a response (jump to another location) based on the student's answers. Or the student input may be included in later examples. Many of the *Gateway Labs* provide such computed dynamic links.

Hypermedia programming environments provide a rich collection of algorithmic building blocks. Where a spreadsheet's building blocks are mostly of the form: "compute a value and store it in this cell," hypermedia programming environments include building blocks of the form: "jump to this location." *Chapter 24: Algorithms Revisited* combines your current knowledge of programming with that of hypermedia environments.

21.5.4

Multimedia

The buzzword *multimedia* actually refers to documents that use several formats or media, for example one that included a motion picture or sound as well as text. In practice, the term often refers to documents that allow substantial user interaction that is,

hypermedia. Most multimedia applications allow two-directional information flow: application to user and user to application. They usually allow references to documents of other forms. For example, many of the *Gateway Labs* allow the user to open a word processing, spreadsheet, or other document. Many multimedia documents can execute animated segments and recorded sounds. Although much more complex multimedia tools exist, most hypermedia applications provide the basics.

21.6

Search Strategies

Internet hypermedia tools open a tremendous world of information. Unfortunately, along with this wealth of information comes new search problems. The task of

3. The best known hypermedia programming environments are *HyperCard* and *SuperCard* (for Macintoshes) and *ToolBook* (for IBM compatibles). One of the newest tools, *Hot Java*, combines *World Wide Web* browsing capabilities with general programming capabilities.

searching for a particular item in such a large information space may seem daunting at first. No single home page can contain an index to every possible topic of interest. The user must systematically search out the desired information.

21.6.1

Standardized Starting Points

Most browsing tools begin at a standard *home page*. Relatively standard features of home pages provide an expectation about what can be found. For example, most organizations have a collection of home pages that reflect the structure of the organization itself. For example, a university typically has a home page, with pointers to the separate home pages for each department; each department's home page points to the home pages for each faculty member within the department. In addition most universities maintain pages for each significant campus-wide activity (e.g., academic computing, college application process, campus social events); each department may have pages for important departmental activities or functions (e.g., departmental graduation requirements, professional clubs, individual courses). As you become familiar with the common organizations, searching becomes easier. A typical home page includes pointers within several dimensions:

subunits of the institution (e.g., each department in the university)

geographically near institutions (e.g., other schools within easy driving distance)

similar departments or institutions (e.g., some other CS departments)

associated topics (e.g., co-authors of the owner of the page)

relevant archives (e.g., a list of all the schools in the state)

the parent in the hierarchy (e.g., the school that contains the department)

local information (nonacademic sites geographically close to the university)

help pages

global collections of indices and search engines (see below)

21.6.2

Superimposed Structure

Arbitrary connections may make hyperspace searches seem complex. Section 21.1.1 provided hints about searching a hypermedia document: every graph contains embedded examples of familiar structures. The embedded tree structure of Figure 21.2 could be used as a guide for searching. Similarly, each tree structure has one or more implicit linear structures. Use these structures to guide your search.

Tables of Pointers.

Suppose you had an electronic version of a textbook. The book of course includes a table of contents and an index. Many documents distribute the table of contents. The starting points contain pointers only to the individual chapters. Each chapter contains pointers to the individual sections, and those sections to the subsections. Following such hierarchies is much like opening folders to find a file. When searching for information, do not search randomly, but follow the general structure provided.

21.6.3

Hill Climbing

Unfortunately, a hierarchical search assumes that you can find a suitable root to use as a starting point. Before you can conduct a hierarchical search, you must get from your own home to a page reasonably close to the target. In the *hill climbing* approach the searcher asks, “Which of these measures will get me closest to my target quickly? (The name, *hill climbing*, comes from the question “which path will get me to the top of the hill quickest?”) Follow that link. The idea is to find an intermediate page related to the ultimate goal in the hope that the intermediate page will contain pointers to the desired subject.

Example 21.1 Finding a Friend’s Home Page. Suppose you are a student in the computer science department at an engineering school in Kansas City and wish to find the home page of a friend a music major at an arts school in St. Louis. When you start your browser you see the home page for your local computer science department. You might get to your friend’s home page through the following path:

Start at home page of the local CS department.

Go to a CS department in St. Louis (follow a similar department pointer).

Go to that school’s home page (upward hierarchical pointer)

Go to your friend’s school’s home page (local school pointer)

Go to the music department (downward hierarchical pointer)

Go to your friend’s home page (downward hierarchical pointer).

At each step you get closer to your friend: your own school, your friend’s school, her department, your friend herself.

21.6.4

Bookmarks and Initial Leaps

A given user is likely to search some locations repeatedly. A business student may frequently look at the *Wall Street Journal* pages. Most hypermedia applications allow users to place *bookmarks* at any particularly useful pages and return to the marked page later. Similarly you may start your search at any known starting point. Perhaps a friend told you about it, or you saw it in a magazine article. Since this does not follow an existing link, it requires knowing the address of the site, a topic which will be discussed in the next chapter.

21.6.5

Understanding Forward and Backward

Unfortunately, some terms such as *backward* are ambiguous. *Backward* may mean “earlier in the document” (e.g., the page that is syntactically before the current page); in other cases, it means “the last page viewed before this one.” Many hypermedia applications distinguish between these two notions of “previous.” Many enable both forms of search. Remember that by the nature of hypermedia, the creator of a document collection cannot know the order in which the reader

will ultimately visit individual pages. Arrows contained as part of a document itself must refer to the syntactic order, or intended order for the document. On the other hand, the browser cannot know the intended order of the document. Forward and backward controls built into the browsing tool will navigate through pages in the order in which you have previously visited them.

21.7

Search Engines

In a small hypermedia document or collection, the easiest way to find a specific item is probably through the table of contents or the index. But by the nature of the Web or *Gopher*-space, no one person built the entire collection. How can anyone index amorphous hypermedia collections such as these? Similarly, how can a user locate the relatively few documents out of the many on the Web that are really relevant? The library user who wants a book on a specific topic goes to the library's card catalog (which today may also be a computer application (see Section 22.2.6)). Powerful programs called *search engines* provide the analogous service on the Web or in *Gopher*-space. A *search engine*⁴ is actually a hypermedia database, providing the user with lists of specific links to information in the Web. Typically, the user asks the search engine to find documents containing a *keyword* or keywords. The query dialog appears very similar to that of a Find command in a word processor or computer system. The search engine returns a dynamically created hypermedia document containing pointers to relevant web pages around the world. Like a database query, most search engines queries can be complex Boolean expressions.

Example 21.2 Using a Search Engine to Get Help on Using the *World Wide Web*. Figure 21.6 shows the result returned by the *Yahoo!* search engine in response to the query to find pages containing the words "beginner" "www" and "help". This particular response found only one document and returned a pointer to the

document itself as well a directory containing that entry. The directory reference might contain other useful documents. Notice that the bottom of the search engine page also includes pointers to other search engines.

Two conventions team up to make the use of search engines especially easy. First, most institutional (e.g., a university or even a department) home pages contain pointers to search engines, making them very easy to get to. In fact many browsers have buttons that take the user directly to a search engine. Second, since a search engine is essentially a database, the user can use familiar tools for searching for finding and selecting items in a database (see Chapter 8 and also Section 18.4.2).

4. As with other applications, all specific search engines provide similar services in similar ways. Individual users will have their preferences but the underlying concepts are identical. Some common search engines include: *Yahoo!*, *WebCrawler*, and *Lycos* on the Web.

Veronica fills a similar role in *Gopher*-space. *Hyperarchive* is a specialized search engine for finding Macintosh applications.



Figure 21.6
Yahoo! Search for “Beginner WWW Help”

How a Search Engine Works.

You can use a search engine without really understanding how it works. However, they represent an example of an important approach to a common problem of computation. Understanding how they work will provide insight into some of the techniques of computer science. A search engine utilizes a special case of the divide-and-conquer approach. Databases cannot make intelligent searches using techniques such as hill climbing, but must use relatively brute-force techniques. Searching all of hyperspace at the time a query is made would require an unacceptably long time. Instead, the search engine breaks the search process into two subparts:

1. Search the Internet during low-demand hours (e.g., middle of the night), building a list of pointers to the pages it finds.
2. In response to a specific user query, search the (relatively small) local database, displaying a list of pointers stored there by the first step.

The search engine actually catalogs items for which no user will ever make a query. But the work was not performed at a *time-critical* point. No one was waiting for the results of the search. Thus

the engine may have performed vast amounts of work, but it did not matter since no one was waiting.

Exercises

21.9 Find your university's home page. Use it as a starting point to find out about your instructor and the department offering this course.

21.10 Find at least one recreational event in a town near you.

21.11 Find information on the Web about the type of computer you are using.

(table continued on next page)

(table continued from previous page)

- 21.12 Use a search engine to find out about your home town. If the town is large, you will want to restrict the search to something smaller. If the town is small, you will want to search for something larger (nearest large town).
- 21.13 Use a search engine to find information about software you have used in this course.

21.8

Summary

Hypermedia is a general term applied to applications that allow the user to navigate through documents in the order of the user's rather than the designer's choosing. A hyper-document contains labeled links from one location to another, which the user can traverse by selecting a button, usually a part of the document itself. Hypermedia offer the combined flexibilities of a sortable database and an index or table of contents.

Important Ideas

hypermedia *World Wide web*
Web

pointer	link	browser
graph	search engine	hill climbing
		home

Credits

Yahoo! is the copyrighted product of Yahoo! Corp.

WebCrawler is operated by America Online, Inc. It is part of the *WebCrawler* project, managed by Brian Pinkerton at the University of Washington.

InfoSeek is a copyrighted service of the InfoSeek Corporation.

Veronica is a service provided through the University of Nevada at Reno.

Netscape Navigator is a copyrighted product of Netscape Communications Corporation.

Gopher was developed by the University of Minnesota.

Mosaic is a copyrighted product of NCSA at the University of Illinois.

Lycos is a copyrighted product of Lycos, Inc.

22

Telecomputing

We shall never be able to remove suspicion and fear as potential causes of war until communication is permitted to flow, free and open, across international boundaries.

HARRY S. TRUMAN

Pointers

Alternate Chapter 21: Graphs and chapter: Hypermedia1

Gateway Networks

Lab:

Lab Unit 14: Extending What You Manual: Know to Traditional Environments

22.0

Overview

The most exciting developments in computer science today are probably in the area of telecomputing. Access to information around the world is changing the way we do business, compute, and even think. What are the implications of using information and computers that are located in other parts of the country or even the world?

22.1

Introduction to Telecommunication

Many terms have been used to refer to aspects of computerized telecommunication. The common buzzword used in the popular

press, *information superhighway*, refers collectively to a future view of telecommunication. At the other end of the spectrum, many old-time network users may still use the term *Telnet*the name of the first U.S. experimental nationwide cooperative network that connected research universities and defense establishments in the early 1970s. Other terms, such as “interactive television,” “bulletin boards,” “news groups,” and “online databases”

1. The material in this chapter and *Chapter 21: Graphs and Hypermedia* are quite intertwined. Local situations, such as availability of software, will dictate the best order for reading this material. The two chapters can be read in either order.

all refer to individual parts of the picture. The common thread among all of these tools is access. Individuals can access resources at institutions around the world. Such pooling of resources makes information tools available to large numbers of users with less total cost. By pooling resources many users can access a single set of data or a single program. For example, the *Wall Street Journal* makes the Dow Jones averages available through “the network.” This means that individuals who need access can check the latest readings whenever they want. They do not have to visit the newsstand to get today’s edition of the *Journal*. More important, the data is available to the customers essentially as soon as it is generated; there is no delay for printing and delivery by truck. Additionally, since the information is electronic, the user can use a personal computer to analyze the information.

22.1.1

Levels of Network

Computer networks come in many sizes. You have already seen networks (in Section 4.2.1), but your experience so far may be limited to local area networks (LANs). Many organizations have larger networks called *wide area networks* or WANs. For example, a large corporation may run a network connecting the machines at all of their sales, research, and manufacturing sites. There is no logical difference between a LAN and a WAN. The only real differences are in the specifics of the technology needed to connect the machines. The most important network is the *Internet*, which is actually a network of networks. For example, a university may maintain its own LAN, but it also maintains a *gateway* or connection to the Internet. Because the machines within the university are all connected to each other on the LAN and the university system is connected to the Internet, the individual machines are indirectly connected to the Internet. Think of it as analogous to your home telephone. Although you receive your local phone calls through a local telephone company, you can also

receive long distance calls through a long distance provider. In effect, that provider connects your local company with another local company, enabling you to speak with a customer of that other local company. The Internet enables you to communicate with computers on many networks.

22.1.2

The “Information Superhighway”

For all practical purposes the Internet is the current equivalent to the services referred to as the *information superhighway*. The term actually refers to a future in which it is envisioned that every house will be connected to a giant computer network providing high-speed delivery of all forms of information from video telephone calls and 500 television channels, to connection with your office and the ability to purchase airline tickets and chocolate. To at least a limited extent, each of these services is a current reality. Certainly not every person has access, and for the most part the service is not fast enough.

The information superhighway may not be a reality yet, but it will be soon. Its

predecessor, the current Internet, is a reality and is now available to most college and university students. The nature of the tools available will certainly change; indeed, new tools and applications appear every year. But each new tool will be related to previously existing tools; each will represent a steady evolution of technology. The following sections illustrate many of the tools available through the Internet as of the date of this writing. For better or worse, that list will probably be out of date by the time you read this.²

Bandwidth.

The problem with the Internet is not actually its speed of delivery. Network communications actually travel between sites at the speed of light. Raw speed is not the problem any more than it is with television or telephone. The limiting factor is what we call *bandwidth*: the amount of information that can be transmitted per unit time. Bandwidth is measured in bytes (or characters) per second. Notice that bandwidth is not a measure of the time required for a message to traverse the distance between two points, but of the time required for an entire message to leave (or enter) a machine. Limited bandwidth does cause a message to take longer because there are many other messages competing for the same pathway. In practice, bandwidth also dictates the number of simultaneous messages that can be transmitted across a network. Just as long distance telephone lines jam up on Mothers' Day, the Internet can jam up when too many users attempt to send simultaneous messages.

22.2

Common Forms of Telecomputing

The term *telecomputing* refers to the relative locations of the computer and the user, rather than the type of application in use. It is not surprising then that many telecomputing applications actually are very closely related to their local-machine cousins. Widespread

use of telecomputing is a relatively new phenomenon and the methods for such use are changing rapidly. Nonetheless, several distinct forms of telecomputing have evolved. At one end of the spectrum is the simple computer access to a remote computer. At the other end are network searches using *daemons* to search for data across the entire world. The following quick overview describes several classes of telecomputing in terms of their most closely related single computer applications.

22.2.1

The World Wide Web

For the casual user, the *World Wide Web* is the single most important Internet tool available. Indeed, since “the Web” can provide a useful interface to most other net applications, it may be the only net application many users will ever need.

2. Once again, we see the ongoing theme of this text: you are learning to use concepts that will be applicable as applications change.

Essentially, the World Wide Web is a very loosely interrelated set of *hypermedia* documents (much more fully described in Section 21.5.1). Each of these documents is located on a machine somewhere on the Internet, but the user need not even know where it is. Instead, the user follows pointers from one document to another. A Web user can access millions of documents in almost any common format.

More importantly for the current description, most *Web browsers* provide interfaces for each of the tools described in the following sections. That is, a user may access tools such as file servers or Email through a Web browser. The Web browser provides a uniform access method for finding and accessing documents of almost any form. If you can access the Web, you can take advantage of the rest of these tools. Even if you will never explicitly use the classes of dedicated applications described below, you will do well to learn a little about them so that you can use the corresponding tools through a Web browser.

If you do not have access to the *World Wide Web*, you can still use the following tools. But in that case, each tool will appear as a separate and distinct application.

22.2.2

Electronic Mail

Internet mail is simply an extension of the local Email you have already used (Section 4.3). A networked user may send Email to any other networked user anywhere in the world. Almost always, the process of sending Internet mail is identical to that used for sending mail locally. Usually the only change is the use of a slightly longer address to specify the location of the recipient. Instead of sending a message to:

user ID,

you send the message to:

`user ID@address`

where *address* is the name of the machine used by the recipient, and “@” is literally just the “at sign” character from the keyboard. The address always follows a simple standard convention, described in Box 22.1, “Understanding Internet Addresses,” which summarizes the universal details of Internet addressing. Not only are the conventions for addressing standard across machines, they are standard across the many Internet tools. Once you learn the address format you will use it over and over.

Delivery of Internet mail does take a little bit more time than on a single machine or local network but not much more. Most messages are delivered within seconds.

The rewards of Internet mail are fantastic. Asking questions becomes a dramatically more powerful problem-solving tool more powerful in proportion to the number of persons of whom you can now ask questions. No longer is the user restricted to local experts. All the experts of the world are potentially available.

Box 22.1 Understanding Internet Addresses

Internet addresses have a standardized format, making them easy to understand. Virtually every U.S. Email address has the form:

user's ID@site name

Site names may appear complicated at first but they really are both simple and standard and all network applications use a single format. Like U.S. postal addresses, the segments of the address are ordered from smallest part to largest. Thus, a typical address might be:

smith@harvard.edu

meaning “a person named ‘smith’ at Harvard University, which is an educational institution.” The only punctuation in an address is the “.” separating the segments. All colleges and universities have the same net designation: `edu`. Other sorts of institutions have other designations, but fortunately there are very few. The three you are most likely to encounter are:

`com`: commercial organizations (e.g., Apple (`apple.com`) or IBM (`ibm.com`))

`gov`: government entities (e.g., the National Science Foundation (`nsf.gov`))

`org`: organizations (especially nonprofit ones) (e.g., National Public Radio (`npr.org`) or the Association for Computing Machinery (`acm.org`))

In addition, some sites are divided into subnets. In this case the name of the specific machine or subnet (called a *domain*) precedes the name of the site. Thus the author of this text is:

scragg@cs.geneseo.edu

or “a person named scragg, in the Computer Science department

at SUNY Geneseo.” Special purpose connections (e.g., *World Wide Web* or file access may be treated as if they were on dedicated machines at the site: www.hp.com or [ftp.nsf.gov](ftp://ftp.nsf.gov)).

The newer international standard is slightly different. Countries other than the U.S. (and some of the newest U.S. locations) use a two-letter designation for organization type (e.g., `co` for commercial), followed by a two-letter designation for country (e.g., `ca` for Canada, `de` for Germany (Deutschland)). For example, the address for the University of Otago, an academic institution in New Zealand is:

`otago.ac.nz.`

Similarly, researchers can collaborate on projects with anyone anywhere with Internet mail. Colleagues can write books, exchange ideas, and update resources even though they may be separated by thousands of miles. Persons need not be separated from the work at their home institution just because they are traveling.

22.2.3

Bulletin Boards and News Groups

Bulletin boards and news groups are much like their physical analog: a bulletin board in a public place, where anyone can place a message. Individuals send messages to an electronic news group and all members of the group receive the message. Such groups are especially advantageous to people in rapidly developing or changing fields. An otherwise isolated individual can now communicate and share information with many like-minded persons. For example, suppose you are the only person at your institution using a specific computer program (for example you want a unique word processor; perhaps you have a specialized need (upside-down letters?) filled by that program). If you have problems or questions, you have no one to talk to. You can use a news group to check in with a worldwide community of users who may have the same or similar questions. News groups make you part of a large team for problem solving.

Although all bulletin boards are conceptually similar tools, the exact mechanism of these services varies widely. The simplest form, a *distribution list*, is a sort of public mail address list. Any mail sent to a specific address is relayed to all members of the list. The list name is actually an alias, like a nickname, for the list of all the members. But it has significant advantages over maintaining your own aliases: it reduces redundant work. Only one person or site needs to maintain the list and all can take advantage of the single list. In fact, most lists are now maintained automatically by *listservers*, a pseudo mail address to which users send messages

saying *subscribe* to place their name on the list (*unsubscribe* serves as the undo).

News groups are a similar, but slightly more formal arrangement: more like a newspaper. Rather than receive a large collection of messages as part of the daily Email, each user accesses a news group through the local *news server*. Each user can subscribe to any number of news groups (meaning she will see those news groups automatically when she uses the server), or she can browse through any group to see the current news for each group (like buying one copy of a newspaper at the store). There are now several thousand news groups that include almost any topic you can imagine from help with specific computer programs to sports events to cooking to news of specific countries.

Many news groups are very large, both in terms of the number of regular readers and the number of items that they carry. Most news browsers help the user weed through the incoming news to find the articles that are of most interest to them. One helpful tool is the *thread*, essentially a title placed on the first article of a series. All related articles then have the same thread. Once a user realizes that she is not interested in a given thread, she can skip all remaining articles.

Typically the user runs a news program on her local machine that automatically makes the connection to the server. The news server itself is a program usually running on a machine within the same local network. News servers reduce the total congestion of news messages over the Internet. Suppose for example, that 100 people at your university belong to the introductory computer science news group. Without a server, 100 separate copies of each news article would be sent to your university, one for each subscriber. Instead, a single copy is sent to the server. Individual local users then get their mail from the local server.

22.2.4

Remote Access

For a variety of reasons, users may wish to run programs on computers other than the ones in their own office or lab. For example, a physicist may want to use a “supercomputer” located at another university to make some very large numeric computations; a researcher may want to look at information held in a database or repository at a distant site; you may be away from home and want to access your own machine. Remote access simply allows you to connect or log on to one machine, using another as your input device. The connection may be through phone lines or dedicated computer networks, but the principle is the same.

Ideally, the program running on the remote machine appears just like any other program to the local user. Perhaps it has its own window. Unfortunately, few such connections provide a wysiwyg interface.³ This means that the user must know the appropriate *command language* for the remote machine. Frequently, remote sites restrict access to certain persons. In such case, you must prove that you are entitled to connect to the remote machine with a *password*. There are as many sets of restrictions on passwords and command languages as there are types of host machines, and as many variations in general interface as there are applications. The

following sections describe a few services that are widely available to the public as special cases of general remote access.

22.2.5

Document Retrieval

Many institutions now provide large, publicly accessible collections of documents. A remote user may connect to these *host* sites and ask for a copy of the document to be sent or *downloaded* to a local or *client* computer. For example, computer companies maintain information about their latest products and reports on problems (and their solutions!); colleges maintain lists of campus activities and admissions requirements; local governments provide travel information for potential visitors. Typically the user connects from a local computer to the host, finds a list of available documents, and issues a request for the ones she wishes to see. The documents are then sent automatically to the local machine. Note that the term

3. One of the most successful attempts to circumvent this restriction is through an interface called *X-windows*.

“document” here is used very generally, and includes not just printable data but other forms of computer information such as complete applications. In general, anything that can be found on a computer disk can be found in document archives. For example, one of the largest archives, maintained at Stanford University, contains several thousand programs that run on Macintosh computers. Similar archives exist for other machines and with other common subjects.

Conceptually, remote access of documents is identical to the use of a file server (see Section 4.2). There are two differences. First, the server may be anywhere on the Internet. More noticeable is that local users will need a separate tool or application to access the remote data. The most common tool is called *FTP*, which is actually the name of the method for accessing the data: *FTP* or *file transfer protocol*. *FTP* allows the remote user to explore directories or folders and access documents much like on a local machine (with a slightly more complicated language but it does accept the all-important command: `help`). Specialized applications such as *Fetch* now provide a wysiwyg interface to the *FTP* program, making *FTP* access almost identical to accessing a local file server.

22.2.6

Database Searches

Many institutions maintain large databasesorganized collections of data, usually together with programs for sifting through it. The most common example today is probably the *online library card catalog*. Many libraries allow users to search for books through their personal computers. Usually these are not restricted to machines located in the library itself, but include any machine which can connect to the library through the Internet. The user runs a program on the host computer for searching the database or card catalog. The host software provides tools specially tuned to the specific database. Thus a library system allows users to search by author,

title, and subject (as well as other approaches). The user does not need any special (local) software for the host machine because the specialized search software is located at the host.

Notice that with databases, the user does not access the raw data directly. That would be cumbersome and time consuming indeed. Many publicly available data sets contain the equivalent of millions of pages of data. Even a small college library may have several hundred thousand books, and the corresponding database will contain as many “cards.” The average user does not want to wade through such large collections of data. This is the advantage of the online database. The search itself is conducted by a *search engine*—essentially a dedicated and specialized database application, like the ones you have used yourself (e.g., those in Section 8.4). The search engine wades through the large collection of data, delivering to the user only the requested items. In fact, most engines will warn the user if a request would deliver too many items to be usable.

In addition to libraries, many organizations and government agencies have specialized search engines enabling users to access large collections of data. These collections may include anything from collections of educational resources (e.g.,

the *ERIC* system) to deadlines and details for grant applications (e.g., NSF's *STIS* system) to the names, phone numbers, and Email addresses of company employees. There are even online "help wanted columns" in which job applicants can find job listings from around the world. *Archie* combines the concepts of database access and FTP access: it is a database containing world-wide listings of publicly available documents at FTP sites. The program *Anarchie* is a search engine for searching *Archie* listings.

22.2.7

Specialized Applications

Although the database is probably the most common, other forms of host programs can be run from remote sites. For example, many universities have programs allowing remote users to find the names, addresses, and phone numbers and other information about faculty and staffeven those without computer accounts. Many governments provide simple little programs called "local time" allowing a remote user to find out the local time in the host country. Perhaps they realize the general lack of geographic knowledge on the part of the largest single group of people with network accessAmericans. Perhaps they just want to make phone calls during business hours easier. Other specialized data servers range from the ridiculous (e.g., the most recently purchased item on a vending machine, or the number of people who have previously read this particular item) to the very useful and hard to find (current exchange rates for any currency or the current stock quotes). The advantage of such servers is that they can provide either very up-to-date (latest weather) or very specialized information (one informs you if the Web browser you are using is the latest versionyou don't even have to provide the details; it checks automatically).

From the user's perspective the distinction between a database and such specialized servers is small or even insignificant. She probably does not care whether the host looked the information up in a table

or computed it on the spot. But the host may invoke any algorithmic strategy for finding the needed information. For example, the *Koblas Currency Converter* can provide the current price of any currency in terms of any other currency. But it does not keep a database containing every possible exchange rate. (Suppose, for example, that there are 100 different currencies. A list of all possible exchange rates would have $10,000 = 100 \times 100$ entries.) Instead, the converter calculates the needed exchange rates by comparing each currency to a single standard. *Chapter 24: Algorithms Revisited* provides an introduction to techniques needed for providing such services.

22.2.8

Network Applications

Finally, there are programs that enable the user to better use other network services. Typically these programs help users navigate through the Internet so they can access other network data sources or even more traditional forms of data. They can access material not just at a single site, but across the entire network. For

example, *Netfind* allows users to find the Email address of a friend or associate even though she may not know what computer the friend uses. The requester simply enters the associate's name and as much other useful information as possible (e.g., name or type of institution, field of expertise) about the associate. The program then commences a systematic search of data at sites fitting the general description. You may think of it as a global generalization of the *finger* or *show-users* program you may have used locally. Such searches involve many machines across the country (or even the world). A single host server requests information from many other hosts, summarizing the results and delivering them to the client user.

Very powerful search engines now exist for searching more general data collections around the Internet. For example, several tools can search for public data sets containing or described by selected keywords. Some popular search engines include *Archie*, *Veronica*, *WebCrawler*, and *Yahoo!*. Although these tools work in different ways and in different domains, each enables the user to locate potentially useful documents at distant net sites.

Exercises

22.1 Laboratory Unit 14 provides examples of most of the types of tools discussed in this section. Follow the instructions in the lab for accessing these tools.

22.2 Find out if your university library has an online catalog. If it does, see if it has this text. Does it have anything else by the same author? If not, find a book by a more well-known author.

22.3 Use a network FTP search tool to find tutorials relevant to

some aspect of
this course.

22.4 Use a net-searching tool to find the Email address of your instructor. Now use the tool to find the author of this text.

22.5 Find a news group related to your hobby. Find another related to this course. Look at the latest entries in each.

22.3 Advantages of Network Access

It is clear that networked information service is or will soon be essential to all areas of business, academics, and science. Information that once required weeks to obtain is now available in seconds; information that was for all practical purposes unavailable is now available; information that once was inconsistent or incomplete can now be held in central locations. Services that once required employment of an outside professional can now be performed in the privacy of your home.

22.3.1

A Familiar Example

Actually most readers are probably already familiar with at least one common form of network-based activity: the automated teller machine. An ATM is not part of the Internet, but the principles are the same. Consider what happens when you use your ATM card. You first *log on* to the local machine (the actual ATM machine) by giving it an identifying card and a password (your *pin* number (*personal identity number*)). Essentially this is the same as a user name and password given to a local computer. The ATM runs only a single program, but that single program allows you to request any of several services:

withdraw money from one account, delivering the cash to you here
deposit money into an account

transfer money from one account to another

check your balance

get a cash advance against your credit card

Usually you can use any of several accounts, and if you have accounts at multiple banks you can get at any account (although that probably requires a different card for each bank). You can request more than one service and request any amount of money (up to a limit) to be moved.

The ATM is connected to a bank network. When you use an ATM, it contacts the parent machine at the bank identified on the ATM. That machine then contacts your own bank's machine, which verifies who you are. The ATM communicates indirectly with your bank and tells it how much money you asked for; your bank checks to verify that you have enough money; and tells the ATM to deliver the money. Both banks update the information about the transaction. General network access is really the same concept

applied to other activities.

22.4

Variations on Network Access

Terms such as the *information superhighway* and *telecomputing* are quite vague and general. Although many tools fall under the general descriptions, the exact nature of the communication can vary widely.

22.4.1

Commercial and Academic Varieties of Networking

Network resources presently fall into two broad categories: commercial services and the Internet. The two categories are fundamentally the same, but differ in philosophy and implementation. Both are now interrelated, but were founded with different philosophies.

Commercial Applications.

You have probably heard of many commercial network servers. *America Online*, *GENie*, *Prodigy*, and *CompuServe* are some of

the best known. With these services you pay a fee which allows you access to a central server. Most servers provide all of the types of services discussed above. The principal advantage of the commercial services is that each provides a central, coherent, and relatively uniform interface to the offered services. Additionally they provide access to services which are essentially commercial in nature. For example, you may purchase anything from theater tickets to stocks and bonds using the commercial services. In addition, anyone with a computer, a telephone, and the money to pay the fee may get access to the network. The fee itself is relatively small; access to a computer and modem is a bigger problem.

The Internet.

In contrast, two of the largest groups of servers on the network are universities and government offices. The most commonly cited advantage of the Internet is claimed to be free is not really true. True, there is no user fee as with the commercial networks, but obviously, someone pays the bills. That someone is the institution that provides the connection. Most colleges and universities feel that the access is worth the price. The individual users at the institution can usually use the network for no additional fee.

The real advantage of the Internet actually comes from its apparent lack of organization. The net itself has no (or little) central coordination and works through the cooperative efforts of the members. Since there is no single central authority deciding what services should be available, no one can prohibit anyone from making a service available. Thus Internet provides access to many services that will never be commercially profitable, and ones which only a few visionaries see as useful. Among the millions of users, there are thousands who have ideas about new services. Compare this to the relative handful of companies making commercial products for computer network use. Sure, many of the ideas will be

worthless, but the volume of new ideas virtually guarantees some great ones. The Internet may be the ultimate in the "better mousetrap" approach to business. The Internet tends to be closer to the cutting edge, but the commercial services provide a more polished interaction.

The distinction between the Internet and the commercial providers is now largely blurred. Users from either can now access at least some of the services of the other. Email flows easily and frequently between them. Most commercial servers provide their customers with a gateway onto the more general net. Many commercial servers now provide database and other services to users on the Internet. Recently many commercial servers have started to provide an intermediate service. For a small fee, these servers make an individual user a node on the Internet.

22.4.2

Means of Access

Telephone Versus Hard-Wired Access.

Computers are connected to networks in either of two principal ways: directly (also known as *hard-wired*) and

via telephone lines. Once connected, there is little to distinguish the two modes with the exception that telephone connections are usually (but not always) slower. It is only in the connection process that the user can observe any difference. With a hard-wired connection, a wire runs from the local machine to other machines on the network. With a phone connection, the user connects the computer to a telephone, using a device called a *modem* (an acronym for *modulator-demodulator*). A modem is nothing more than a device to transform the internal form (digital) of the computer's outgoing messages into the internal form (analog) used by a telephone, and perform the reverse transformation on incoming messages. Telephone access effectively takes advantage of an existing network (the telephone system) to connect any number of individual machines to other machines or networks of machines.

Net Versus Machine Access.

The telephone versus hard-wired access is easily confused with a second distinction. A remote user can log directly onto a host machine. This connection can be either hard-wired or via a telephone line. In either case, the user's local machine behaves like a terminal to the single host computer. A computer can also access the Internet by either method: telephone or hard-wired. In this case the user's machine becomes a machine (*node*) of the Internet. With such a connection, the user can use any Internet tools.

22.5

Ethical Considerations of Telecomputing

The fantastic advances in the field of electronic communication constitute a greater danger to the privacy of the individual.

EARL WARREN

22.5.1

Freeware Versus Shareware

Some users can be genuinely confused by the descriptions of two

classes of software found on network archives: *freeware* and *shareware*, which entail related but different distribution techniques. Understanding the differences is easier if you first understand the similarities. Developers of computer software tend to be enthusiastic users of computer tools. They realize that some computer tools create an easier way to distribute software. Files can be transferred across computer networks in much the same manner as they are transferred from one disk to another (see *Chapter 3: Documents as Objects*). Some software developers prefer to sell or distribute their products through computing networks rather than sell them in stores.

Whether an author uses freeware and shareware depends on what she expects for her efforts. *Freeware* is exactly what it sounds like: the developer is a seemingly magnanimous person who wishes to give the software away. This

giveaway may be out of the goodness of her heart, or she may have some other motive such as establishing name recognition, or a product base for future sales.⁴

Shareware, on the other hand, is for sale. The developer wishes to sell the product. Rather than shrink wrapping the product and placing it in a retail store, she makes it available electronically. She expects and is entitled to compensation for her work. Until about 20 years ago, street corner newspaper racks were not enclosed. Buyers put a dime in a slot when they took a paper. There was no physical constraint to make them do so; they did it out of honesty. Some charities still use similar arrangements for selling candy on restaurant counters. Shareware uses exactly the same principle: users are expected to pay, even though the seller has no physical security guards in place.

Shareware has advantages (over commercial distribution) to both seller and buyer. For the seller, it is easier: there is no need to find a distributor and no need to package the product. It is also much faster: the product can be available as soon as development is complete. Traditional commercial products require months while the package is designed, disks are copied, and so on. For the buyer, the product is much cheaper a direct result of the money the developer saved by using the shareware approach. Perhaps even more important is the test drive capabilities. Part of the philosophy of shareware is that the user should be able to try a product first and then decide if she wishes to use it. Such test driving falls within the shareware guidelines.

A third variation, *postcardware*, falls between freeware and shareware. Postcardware is shareware with a price of “one postcard.” Thinking about postcardware helps clarify the distinction between freeware and shareware. The author of postcardware is effectively saying, “I am basically giving this software away (like freeware). But since it is my creation, I am entitled to ask for

something (like shareware). So, I ask only that you send me a postcard as your payment.” With postcardware, users get something essentially for free, but the author also gets something: the satisfaction of knowing that someone likes her product.

Two more aspects of software distribution confuse many new users. First, even though freeware is free, it is usually copyrighted. The copyright holder owns the product. Since the developer holds the copyright, she may place restrictions on its use. Common restrictions are:

The product may only be used in its entirety.

The product may not be sold for financial gain.

The copyright notice must be included.

4. The most famous example of this technique: Gillette gave away safety razors in order to sell the blades. For every razor they gave away, they sold hundreds of blades to use with it. (King Camp Gillette would have been quite upset if persons infringing on his patent reduced the number of blades he could sell.)

These are very reasonable restrictions from the developer's perspective. How would you like part of your term paper (perhaps a conclusion with the supporting arguments) distributed as if it were the entire paper? How would you like to work hard on a project and have someone else make the money?

Finally, even though a product is copyrighted and for sale, the developer may encourage users to *distribute* it by making copies for their friends and associates. The key word is *distribute*. Recall that the developer does not have a retail distributor or an advertising campaign. Both the advertising and distribution mechanisms are like word of mouth advertising. You are free to distribute shareware, even to use it long enough to decide if you like it. You only have to pay if you decide to keep and use it. It is in the interest of both the author and future users to distribute the product. It is not in the author's interest for you to use it unless you pay.

22.5.2

Breaking and Entering

Access is a two-way street. A door from a computer onto the network can also serve as a door from the network into that computer. Without additional safeguards, network access can create a nightmare of unauthorized access to your computer, your data, and anything that your machine gives you access to. Fortunately, virtually all machines have some sort of protective software to reduce such unauthorized tampering. Unfortunately that software may not be foolproof. No matter what precautions you take (short of disconnecting your own access to the network), a dedicated snooper with a reasonable amount of expertise may be able to get into your machine.

Some of these snoopers present the argument that it is each user's responsibility to safeguard her own machine, and any machine left unprotected (or inadequately protected) is fair game. They feel they have a right to invade any such machines. Once again, the "would

you do it without a computer?" analogy casts a very different light. If I forget to lock the front door to my house, does that give you the right to enter, take what you want, or damage my possessions? Virtually everyone would say nothe same is true of unauthorized computer access.

But I Didn't Take Anything.

Some computer snoopers claim they were only looking around or trying to see if they could do it. Again: would you accept that explanation from an intruder in your house?

Confusing Terms.

The words commonly used to describe such intruders have confusing sounds, histories, and meanings. The oldest term, *hacker*, at one time referred to someone who hacked her code together, not elegantly, but she got the job done. Later the term evolved into one who really understood the minutia of the machines, and finally into someone who broke into other machines. *Cracker* is a play on *hacker* and is applied to the more malicious offenders.

Exercises

- 22.6 Are there any advantages that shareware brings to society (not just the authors of products)?
- 22.7 List possible arguments in favor of hackers. For each argument, write a paragraph attacking that argument.
- 22.8 Team up with a classmate. Each of you try to break into the other's machine. Hints: You should each connect yourself to the network as if you plan to access other machines. Use those tools to find and access the other machine. Note that this task may turn out to be easy or hard depending on many local factors. Remember to do this only with a cooperating friend.
- 22.9 If your teammate in Exercise 22.8 succeeded in getting into your machine, how did you feel about it?

22.6

Summary

The Internet provides access to remote sites around the world. This access includes a wide variety of services such as Email, file services, databases, library and information servers, search engines, and remote sharing of software. The most convenient method of access to the Internet is through the World Wide Web. Other access paths include both alternative interfaces (e.g., specialized database programs) and alternative services (e.g., commercial providers).

Important Ideas

shareware Internet

Email modem

database	intellectual property
remote access	file transfer protocol
freeware	World Wide Web
news	search engine

Credits

America Online is copyrighted by America Online, Inc.

GENie is a copyrighted service of GE Information Services.

CompuServe is copyrighted by *CompuServe Inc.*

Fetch is copyrighted by the Trustees of Dartmouth College.

X-windows was originally developed at MIT; commercial vendors have since made X an industry standard for UNIX platforms.

Koblas Currency Converter is the creation of David Koblas and provided through *Global Network Navigator*.

Archie was developed at McGill University Computing Center, and is now supported by Bunyip Information Systems Inc.

Netfind is supported by the University of Colorado.

Anarchie is copyrighted shareware created by Peter N. Lewis.

Prodigy is copyrighted by Prodigy Services Company.

23

Hypermedia Construction

The art of progress is to preserve order amid change, and to preserve change amid order. Life refuses to be embalmed alive.

ALFRED NORTH WHITEHEAD¹

Pointers

*Gateway*Hypermedia navigation

Lab:

Lab Unit 15: Hypermedia

Manual: Construction and the World

Wide Web

Unit 16: Hypermedia

Construction with Dedicated
Tools

23.0

Overview

As with other living documents, the designer and the user or reader of a hypermedia document may be different individuals. Chapter 21 discussed hypermedia from the perspective of the user. From the perspective of the designer, the corresponding issues take on slightly different forms. For any hypermedia document, the designer must tell the application how to find new locations when the user follows a link. As with programming, the process includes more than just the mechanics of instructing the computer. The designer of a hypermedia document must also assure that the document is well organized and that the visual appearance of the document reflects that organization.

23.1

Dynamic Links

The construction of hypermedia is a surprisingly easy and straightforward task. A hypertext document need be no more than a simple word-processed document with the addition of dynamic links. The only new task for the designer is to build these

1. Whitehead (1861-1947) was a British philosopher who co-authored one of Bertrand Russell's most important works, *Principia Mathematica* (see Box 14.1).

links. The concept is easily illustrated with a relatively simple example: a document with a dynamic table of contents. In this document, selecting a chapter title will be equivalent to moving the user to that chapter. Suppose that the document is already stored in an electronic format (so far no problem: this just means it was created with a word processor or other appropriate tool). Further suppose that the first page is a traditional table of contents listing the remaining sections. The table includes the starting page for the section. That is, it points to each section. The only missing detail is the mechanism for requesting that the application “move to the indicated page.” As with other living documents, the document designer and the document user or reader may be different individuals. In this case the construction of a link and following a link are two separate activities. The user sees links as buttons that change the current focal point to a new location within the document. The document designer must create that link.

23.1.1

Construction of a Link

A dynamic pointer requires three critical elements:

a pointer to a new location in the text,

a *button* or *selector* for the link, and

an application capable of making the dynamic link.

By definition, all hypermedia applications satisfy the last requirement. The remaining two parts are simple but intertwined. To create a full table of contents, the designer simply repeats the two steps for each entry in the table. That is, the designer attaches a dynamic link to each line of the table of contents.

23.1.2

One Task, Two Tools

Although the logical process of creating a link is the same for all

environments, the exact process of creating this dynamic pointer depends on the specific environment. There are two classes of hypermedia applications, which for purposes of this discussion can be called:

compilation applications, and

interpretation applications.²

The obvious differences between these two types of tool are related to the accessibility of the links themselves. Compilation applications include commands that create links automatically. The designer requests a link and provides the needed information; the application converts that into the needed link. The user

2. The terms *compilation* and *interpretation* are actually borrowed from higher-level languages. Their use here for hypermedia documents is nonstandard, but this usage does capture much of the original distinction. A compiled higher-level language is translated from the form written by the designer into an optimized internal representation. An interpreted language remains as the designer wrote it until it is actually used.

can then tell the application, “follow a link,” but cannot see the details of the link itself. With interpretation tools, the designer describes the link in the same way she completes the document itself. The application does not build a link at that time. The end user looking at the document can see either the description of the link or ask the application to interpret (follow) the link.

23.1.3

Compilation Applications

Compilation applications include dedicated hypermedia creation tools such as *HyperCard* and *ToolBook*. The application provides all the needed expertise for creating links. Generally the application assumes that the designer has immediate access to the entire hypermedia document. That is, they are used best when creating the entire document rather than a single page that fits into a larger collection. These applications provide tools of the form:

Give me a link.

From the document designer’s perspective, she asks for a link, and the application responds by asking for details:

What should the link look like visually? (button, text, location, etc.), and

Where should the link take the user?

The designer simply provides the details, usually by selecting them from a menu. The dialog for the table of contents example might look like:

Designer: Create a new link. (perhaps using a menu command)

*Application: What should it look like?
(a button, highlighted text, a picture?)*

Designer: Highlight the text.

Application: Where do you want the button?

*Designer: Indicates a point on the page.
(perhaps by dragging the button there)*

*Application: Moves the image to the correct location and then
Asks “where should the link point?”*

*Designer: Selects a page.
(perhaps by navigating the document until the
page is
reached).*

*Application: Create the link.
(which is not actually visible to the designer).*

The designer then tests the link and verifies that selecting the button does indeed take the user to the desired page. Once constructed, the internal representation of the link is invisible. The designer need not be aware of any internal representation for the link, but does need sufficient knowledge of the structure of the document to navigate to the desired target page before constructing the link. The user will never see them.

23.1.4

Interpretation Applications

In the interpretation process, the designer and user may actually use different tools. The designer uses a general data-processing tool such as a word processor. Using that tool she provides a formal description of the link. The hyper-document contains only a description of the link. The user uses a hypermedia interpretation tool such as *Netscape* to follow the links. That application translates the description into the needed form when the link is actually followed. For example, most World Wide Web navigation tools are really user tools; the designer uses a common word processor. If the original table of contents looks like:

<i>Chapter 1</i>	<i>page 2</i>
<i>Chapter 2</i>	<i>page 10</i>
<i>Chapter 3</i>	<i>page 20</i>

the designer simply uses a word processor to insert phrases into the table of contents, indicating that the entries are actually pointers. The results are logically equivalent to text of the form:

Pointer:

looks like: "Chapter 1 page 2"

linked to: address or location of chapter 1.

Pointer:

looks like: "Chapter 2 page 10"

linked to: address or location of chapter 2.

Pointer:

looks like: "Chapter 3 page 20"

linked to: address or location of chapter 3.

That is, the designer writes and sees only formal descriptions of the pointers. From the designer's perspective, the pointer is a slightly corrupted version of the original text. The user, using an end-user viewing tool, sees the sections of the pointers labeled "looks like:"

In this case it is the original table of contents, with each entry highlighted. When the user selects an entry, the navigation tool jumps to the associated page. If the user wished, she could also see the original description. Notice that interpretation tools place a lot more demand on the designer: she must describe the link without being prompted and must be able to envision the final result she has described. On the other hand, the document can be interpreted by any application that knows the code. The designer can create links to pages of any location or document, if she knows its name: she need not be able to find the document at the time of the design.

In practice, the syntax of the description for interpreted languages is more arcane than the above suggests. For example, World Wide Web documents are described in the *hypertext mark-up language* or *html*. In this language a table of contents entry might look something like:

```
<A HREF = "machine-and-
directory holding the book">Chapter 1</A>
```

Dissecting this command reveals a structure essentially identical to that described above. Each complete link description is called an *anchor* (suggesting that the two locations are anchored together) and delimited as:

```
<A . . .>. . .</A>
```

Notice how the beginning and ending delimiters are closely related in this notation. The body of the link has two parts, the “link to” or dynamic hypertext reference (HREF)

HREF = “*machine-and-directory holding the book*”

and the “looks like” or visible display portion:

Chapter 1

The first part names the exact location. The second shows how it appears to the reader. The designer must describe this information to the application. For example the location description:

```
href = “http://www.foo.edu”
```

indicates the home hypertext or World Wide Web (WWW) document located on the server at an educational institution called Foo. Notice that the form of the address is identical to that of Internet Email addresses (See Box 22.1).

Three important advances enable even a beginner to get past the ugly surface appearance and create functional hypermedia documents. First, several menu-oriented design tools now exist, enabling users to build much of the document without first memorizing all of the syntax. These tools provide menus of legal link forms, allowing the user to select whichever is appropriate. In appearance these tools may resemble the compiled applications but their output is a set of link descriptions.

Second, the syntax of basic entries is very standard. The user can follow a very standard and repetitive algorithm to replace each

original text entry, such as

Chapter 1

with the hypermedia entry

```
<A HREF = "machine-and-directory holding the book">Chapter 1</A>
```

Only the single segment "*machine-and-directory holding the book*" needs to be created from scratch. The "glue" parts

```
<A HREF = ". . ." . . .>
```

are standard for all entries.

Finally, all navigation tools now display the address of each page along with the content of that page. Thus, a user can find a page using a navigation tool, copy the page address, and insert that address into her own hypertext document. Laboratory Units 15 and 16 detail the process for specific hypermedia applications.

Exercises

- 23.1 Select any *Gateway lab*. List as many links as you can find in the lab.
- 23.2 Start on any *World Wide Web* document and list all of the documents that you can get to within three steps.
- 23.3 Would you describe the hypermedia tool you have used in your class as compiled or interactive?

23.2

Visualizing Data Transitions

Although the syntactic details for creating links varies with the tool, the logic for designing the hypermedia document does not. The designer needs to think through how the document will be used and what links will be needed. For example, a user should be able to navigate the document in orders other than the default. A user who starts a chapter and then realizes that it is not the right one needs to find the appropriate chapter, perhaps by first returning to the table of contents and then selecting a new chapter. That implies a need for an easy way to return to the table of contents. In general, every document will need several forms of link. The designer should consider these links before creating the document and she should do so in a top-down manner. How will the document be used? What links will that require? How else will it be used? What if the user makes an error? Visualizing the types of links and their uses will greatly simplify the process.

23.2.1

Modeling the Document

Generally, the designer should first visualize the entire document as a graph. Then she should envision appropriate subgraphs in the form of common data organizations such as linear or hierarchical.

The segments of the graph should contain standard pointers that reflect these structures. Finally, each graph should contain *ad hoc* structures that move the user to arbitrary locations.

Multidimensional documents are harder to envision than simple documents. The designer should make an extra effort to visualize the structure of the document before building it. In the top-down design process this selection comes just before the outlining step. In effect the designer must decide how many (and what) dimensions the document will have. Usually the best way to envision the document structure is to draw a diagram showing the major links. For example, Figure 23.1 depicts the major links used in a typical *Gateway lab*. The figure does not show all the links in the document, just the major classes of link. Even so, the diagram can easily become complicated and cluttered. Building the diagram in advance will help the designer recognize special circumstances, organize the document, and avoid clutter. The world does not end if you later discover a missing, but necessary, linkbut it does complicate the task. Unplanned links:

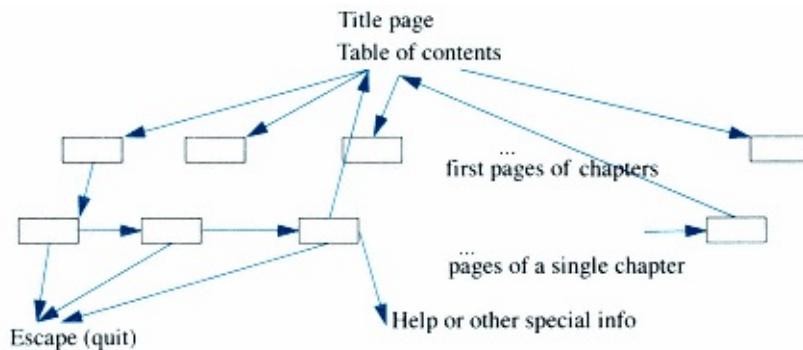


Figure 23.1
Outline of “Simple” Hyper-document

clutter the individual pages as you add new buttons
force you to redesign previously completed pages
often do not take advantage of generalized structure. Although you can usually copy a single link structure for use on several pages, an unplanned link may be added too late to take advantage of such savings.

Exercises

23.4 With the *Gateway lab* you used for Exercise 23.1, draw in the links.

23.5 Draw the structure for the directory of files on your disk. Include extra hyperlinks that point to related items. For example, each of the individual items in a folder called `letters` might have an additional link to the application with which they were created.

23.3 Logical Types of Links

The primary organization for some data sets is obvious, as are the required sets of links. For example, a book has chapters and will need table-of-contents and index links. It may also need links for

illustrations, tables, authors that are referenced, or other special indexes. A computing book may have a separate index for programming terms. A term paper is probably similar to a book. The principal organization for other data collections may not be so obvious.

23.3.1

Linear Structures

Forward Movement.

Any document or section of a document that is essentially linear (e.g., as described in Section 5.1) should include forward links a way to get to the next section, much like turning a page or skipping ahead

to the next section. This link by itself will enable a user to read an entire document, but used alone, it lacks the flexibility characteristic of hypermedia: the reader may only read the document through from front to back, much like scrolling through text. Usually, a forward step moves one “visual unit,” such as one page of text or one screenful of material.

Move Back.

Backward movements are a sort of undo or rewind: “Move back to the last section. I want to reread it.” Although not invoked nearly as often as the forward move, this capability is essential even for an essentially one-dimensional text document.

23.3.2

Hierarchical Structures

Most documents have a hierarchical (see Section 6.2) as well as a linear representation. At a minimum, they can usually be represented as a document with a table of contents.

Down One Level.

Every hierarchical document requires links to the child nodes.

Up One (or More) Levels.

In fact, the user may want to move to any location in the entire document. Although it is not possible for the system designer to anticipate every possible jump, it is essential that every jump be possible. Unfortunately, including separate buttons for every possible move would fill up the page, leaving no room for real information. The “return to table of contents” link reduces these problems. A user can get to any location in two steps via the table of contents:

Return to table of contents.

Jump to named section.

Notice that a move to the table of contents is a move logically upward in the hierarchy, perhaps to the root of the tree.

23.3.3

Ad Hoc Pointers

For some hypertext documents, the notion of forward may not be particularly useful or meaningful. Hierarchical and linear pointers may both be insufficient. *Ad hoc* structures (see Section 5.3.1) allow potentially any link. If a user really has complete freedom to go to any new location, there may not be an obvious “next” location. For example, in a collection of recipes, the notion of “next” is not particularly obvious. Some flavors of *ad hoc* links are especially common.

Help.

Help links take the user to extra material that is not in the mainstream of the text flow. The user only sees this material if it is needed. Help is much like a reference in a text of the form “for more information see. . .”

Choice of Scenarios.

One major advantage of hypermedia is that the user can select the order of material or even the relevant material. A document could have two examples. The user could ask to see the more relevant of the two. For example, a document could describe two different word processors, and the user could ask for the one appropriate in the local domain. A *hyperstory* allows the reader to select what happens next. Some very elaborate hyperstories now exist in which the user interacts to create any number of possible stories.

Selected Topic.

Often a reader of a document encounters an unfamiliar word, term, or expression. The designer can often predict these terms. For example, it is easy to forget terms that were mentioned only briefly in an earlier chapter of a textbook. When a user encounters these words later, it would be nice at that point to say, “What was the definition of that word again?” Hypermedia documents can allow the user to select the forgotten word and immediately find the definition of the term.

Exit.

At any point the user can quit. This really is not a link in the usual sense; it does not take the user to another spot on the document, but quits the application or moves out of the current data collection. From the user’s perspective, getting out may require the same action as moving to a different part of the document.

Exercises

23.6 For the *Gateway lab* used in Exercise 23.1 and 23.4, categorize each according to the structure that it represents.

23.7 Start with the home web page for your institution and categorize each link you find on that page according to the

structure that it represents.

23.8 Describe *Gopher* documents in terms of their structure.

23.4

Guidelines for Creating Links

The default order of most paper documents is clear: the user continues sequentially until reaching the end. In a hypertext document the user may navigate in any of several orders, but it is up to the creator of the document to ensure that all appropriate links are available. The links needed will vary from document to document, but several general principles help assure that the needed links are present.

Consider Basic Organization.

Many data collections have an intrinsic data organization such as hierarchical or matrix. Be sure to allow the user to follow the natural paths. Do not force this organization on the user, but make the natural paths easy to follow.

Assume a Naive User.

Links should be explicit and not rely on an understanding of details of either the viewing application or the subject matter. All links should be built in. Do not assume the user knows application commands or control keys for moving to the next page of the document. In compiled applications, avoid the “no exit” problemdocuments that do not let the user quit the application.

Avoid Dead Ends.

This requirement sounds obvious, but it is frequently missed. Every page of a hypermedia document must have at least one link to another page. The link may be as simple as “go to the next page,” but it must exist. Otherwise, a user who reaches that page is stuck there.

Avoid Orphans.

An orphan is a page which cannot be reached. Every page must have at least one path that leads to it. Again this seems obvious, but it is easy to forget, particularly when editing an existing document. For example, suppose some given page has only one path leading to it. If, during editing, the designer removed that page, no path would remain. Similarly, a designer using shortcuts (e.g., as described in *Chapter 19: Abstraction, Abbreviation, and Macros*) to navigate the document might forget to add the explicit user-oriented link.

Allow Escape Paths.

A user who discovers that the current section of a document is not useful needs a path out of the section or out of the program. For example, an experienced user may want to skip an entire introductory section. There should always be a way around material the user does not want to see. As a default, the table of contents or “home” links provide a relatively simple path to new locations, as does the “forward to next section” link. The “quit” command can

serve this purpose, but it should be absolutely the last resort. Failure to follow this rule is actually the primary problem causing the complaints about voice mail systems mentioned in Section 21.2.3.

Be Flexible.

Generally, allow the user to select a path through the material. Do not force the user to follow the designer's concept of the best order. Flexibility of order is the defining characteristic of hypermedia. Failure to allow this flexibility removes the modifier "hyper" from the document. Escapes should not force the user to repeat long sections, or traverse circuitous paths. For example, if the only way to get back to an earlier page is to quit and start over, the user will be reluctant to return to review material. Remember: the user wants to get on to the new path as quickly as possible.

User Level.

Remember that not all users have the same background. Users with various knowledge levels will look at the document. Some users will want a method for asking for extra help. Others will want a way to skip over simple material. Provide paths for the skilled and for the unskilled. Also try to accommodate multiple ways of viewing a problem. For example, some people

want an example right away while others want a complete definition of all terms first.

Provide Cross References.

Cross references are the heart of hypermedia. Provide frequent links to related material. Allow the user to find new information as she needs it.

23.5

Visual Link Conventions

In addition to logical layout the designer should pay attention to the physical appearance of the document.

23.5.1

Buttons and Structure

The traditional representations for links reflect the assumed structure.

Forward and Backward Links.

The symbol for a forward move through the document is a simple right-pointing arrow (see Figure 23.2). Sometimes single and double arrows are used to distinguish small-grained steps (e.g., next page) and larger-grained steps (e.g., next chapter). Since the forward step usually provides a primary path through the document, it should be located in a prominent location, such as a corner.

Backward pointers are usually created analogously to forward pointers: a left arrow indicates a backward step. A move back to a previous section is a larger or double arrow to the left.

Hierarchical Links.

Hierarchical structures use up and down to indicate up and down the hierarchy. An up-arrow or a bent arrow usually means “move up a logical level” (see Figure 23.3 on page 412). Since a node may have several children, “down” is not clearly represented by a simple

down arrow. Downward links are usually represented by the name of the child sections (e.g., chapter names or numbers).

Other Standard Conventions.

Additional rules of thumb can help the hypermedia designer build useful *ad hoc* structures. Rules of thumb include both generally good practices and rules whose effectiveness is derived from common usage. Even though the rule may not be intrinsically obvious, it can become

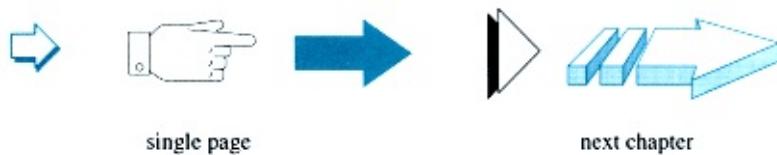


Figure 23.2
Forward Arrows



Figure 23.3
Common “Move Up a Level” Link Symbols

obvious through frequent usage. For example, a red octagonal road sign has no intrinsic meaning, yet every motorist will recognize its conventional meaning. Similar conventions for button design help provide more legible hypermedia documents. Make the design consistent with standard usage if possible. Just as the user benefits from standard menu designs, she will benefit from standard buttons. Use nonstandard buttons only if you have an explicit reason for doing so.

Labeled Buttons.

A button can have an arbitrary label. The label should correspond to the logical link (e.g., the topic name, not the page number). The button label should suggest its destination. It should obviously be a button (in some applications this will be automatic).

Keywords in Text.

Blocks of text may contain references to topics described more fully elsewhere. By convention, the word itself can serve as a button. Obviously, any word used as a link button must be distinguished in some manner (e.g., bold font). For example, a text could underline all words that serve as hyperlinks.

Pictures as Pointers.

If a page contains pictures, individual segments of the pictures can be used like keywords in text. A user who selects a picture part would move to a section describing that part. For example, a medical text might contain a picture of a skeleton. Selecting the foot of the skeleton takes the viewer to a detailed picture of the

foot. Selection of the fifth metatarsal takes the user to a description of that particular bone. Buttons of this sort are especially useful in situations where the actual names of objects may not be readily available to the user.

Obvious Buttons.

Buttons should be easily recognized. They should be easily distinguishable from blocks of texts and pictures, that do not serve as buttons. One common convention is to use buttons that look like physical buttons, such as those on the left of Figure 23.4.

Consistency.

Be consistent in your choice of button design. Consistency is probably more important than the actual choice of button design. Once a user sees a few examples, the pattern becomes clear. Pick a design for each logical button

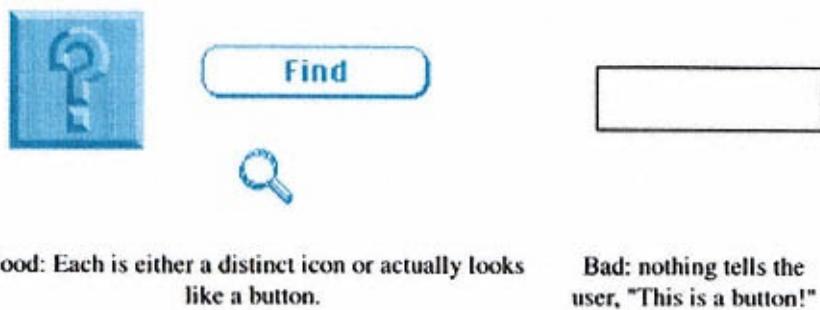


Figure 23.4
Button Design

type and stick to the design throughout the document. Use a single design for all forward steps, and another for all backward steps, and so on. Similarly, use a single convention for any in-text buttons.

23.5.2

Placement of Buttons

An obvious potential problem with a hypermedia document is clutter. Consider where and how hyperlink buttons are best placed to avoid confusion. Increasing the information contained on a button also increases the visual complexity of the document. Try to place the button consistently from page to page, so the user need not search for the button.

23.5.3

Variations on Simple Links

Some hypermedia tools allow the designer to create special effects. While these do not change the logical power or expressiveness of the document, they can serve to add variety or even to draw the user's attention to specific locations. At present these are more common in the compiled documents.

Timed Links.

Many situations seem to require multiple links or links that only last for a short time. For example, many authors want to provide credits the author, illustrator, copyright owner, dedication, publisher,

and so on but (much as we authors hate to admit it) the user would usually prefer to skip over this information and get on to the next item. A link to a page showing credits could automatically terminate and proceed to new material after a set amount of time.

Similarly, a sequence of pages, each appearing for several seconds, can provide a professional-looking introduction to a document. For example, the title page, credits, an overview and two pictures may each appear on the screen at start-up time. Or a “just a moment please” page could appear during that agonizing time while the user waits for the entire document to be loaded into the machine. Finally, a rapid-fire series of pictures could simulate an old-fashioned

movie. If each picture is just slightly different than the preceding picture, viewing them in rapid succession generates a sensation of motion.

Do not use timed links to lead a reader through a document. They should be used for effect, not control. A timed link can easily cause a reader to miss a section. Any reader who does not read at the same rate that you read will find this very uncomfortable. In most situations, let the user decide how long to look at a page.

Special Effects.

In some situations, special effects can emphasize the transition between two pages. For example, if the pages are very similar, a user who happens to look away for a second may not realize the page has changed. Special effects such as blinking the screen, sliding one picture off to the right while a new one slides in from the left, or dissolving the old picture so the new one seems to appear behind it all serve to ease the transitions for some situations.

Similarly, a sound can attract the user's attention. Sounds are particularly useful for indicating errors or the completion of an action that requires an unusually long time. For example, if a transition requires 15 seconds while a new segment appears, the user may get bored; a sound will bring her attention back to the screen.

Pictorial Information.

Use pictures or illustrations wherever they increase the understandability of your document (see Sections 7.6 and 21.4.3). Since hypermedia documents are living documents, this is often very easy. For example, you may be able to point to existing documents without actually copying them.

Exercises

- 23.9 Build a hypermedia document with one page describing each of your classes. Allow links from each class to the one that follows it chronologically and the one that follows it in course-number sequence. Add a “table of contents” allowing you to get to any class.
- 23.10 Build a simple “*Dungeons and Dragons*” game using hypermedia. In *Dungeons and Dragons*, the player moves around a maze of rooms (the dungeon). When the user goes to a door, the scene changes to the next room. In each room new problems befall the player.

23.6 Problems of Hypermedia

A document designer must always select the best tool for the job. Like all other applications, hypermedia tools have their strengths and weaknesses. Some of the more problematic situations within most existing hypermedia tools include unexpected changes and failure to consider the user’s needs.

Changes.

Compared to other living documents such as databases, it can be relatively difficult to add new items to a hypertext document.

Adding a new page can require building several new links. If the links are absolute rather than relative, this can take more time than adding the information itself. For example, when adding a new page between pages 7 and 8, the designer must assure that the next page link on page 7 goes to that new page, not to the old page 8.

Human Factors.

The flexibility of hypertext also allows for a potential failing when used by inexperienced designers. The final product will be only as good as the designer. The flexibility allows for additional situations in which the designer can confound the user. Dead-end paths, or other “you can’t get there from here” situations will make an application seem useless. In a simple word-processing document, the user can find any item by reading the entire document; in hypertext the user may not be able to find the segment to read.

23.7

Another “Hyper”:

Hypercubes

The prefix *hyper* appears in many contexts throughout the various science fields. Some are particularly relevant to computer science. Mathematicians often seek generalizations that enable them to use one simple description to describe large groups of (seemingly disparate) objects. This creates odd-sounding words like *hyperspace* and *hypercube*. In the case of cubes, they noticed that in a standard three-dimensional cube, each corner or *vertex* is connected directly to three other vertices. A three-dimensional cube has 8 (=2³) vertices and 8 (=2³) edges. A square is a two-dimensional hypercube with 4 (=2²) vertices and 4 (=2²) edges.

Similarly, a line and a point are one- and no-dimensional hypercubes, respectively. Interestingly, any hypercube of dimension n can be created from two cubes of dimension $(n - 1)$. For example, a three-dimensional cube can be created by first creating two squares (two-dimensional cubes) and then connecting each vertex of one square to the corresponding corner of the other (see Figure 23.5). Similarly, a two-dimensional cube (square) can be built from 2 one-dimensional cubes (lines). The analogy can even be carried to construction of one-dimensional cubes (lines) from zero-dimensional points. The interesting question then is how to create a

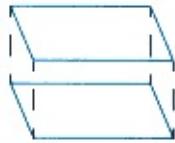


Figure 23.5
Building a Cube
from Two Squares

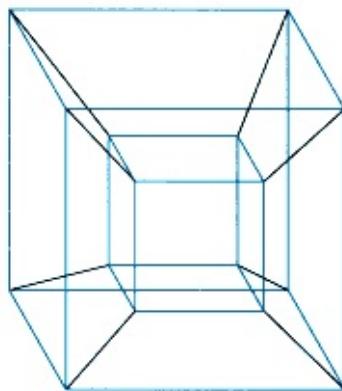


Figure 23.6
A Four-Dimensional Hypercube

four-dimensional hypercube. Following the above guidelines, one should be able to build it by first building two three-dimensional cubes and connecting all the vertices.

Unfortunately, most humans cannot visualize objects in four dimensions. The drawing in Figure 23.6 is only two-dimensional, but it depicts a three-dimensional visualization of the four-dimensional cube. It is constructed of two cubes (an outer and an inner). Each vertex of the outer cube is connected to the corresponding vertex of the inner cube. It has $16 = 24$ edges and vertices. Each vertex is connected to four others.

The hyperspace concept appears throughout science fiction as a way of visualizing the time-space continuum. Hypercubes, in particular, are important to computer scientists because they help us visualize a way to network several computers together efficiently: just follow the edges of an imaginary hypercube.

Exercises

- 23.11 Label the vertices in the hypercube in Figure 23.6. Select any two nodes and find a path between them traversing no more than three edges. Repeat the process until you are convinced that it can always be done.

23.12 In general how many steps are required to link any two points in an N -dimensional hypercube?

23.8

Summary

Hypermedia is a general term applied to applications that allow the user to navigate through documents in orders of the user's rather than the designer's

choosing. A hyper-document contains labeled links from one location to another, which the user can traverse by selecting a button. Usually the button appears to be part of the document. Hypermedia offer the combined flexibilities of a sortable database and an index or table of contents.

Important Ideas

button hypermedia multimedia

graph embedded structure
compile

interpret

24 Algorithms Revisited

The mind ought sometimes to be diverted, that it may return the better to thinking.

PHAEDRUS

Pointers

*Gateway Algorithms in
Lab: Hypermedia*

Lab Unit 17: Creating Algorithms
Manual: Using Hypermedia Tools

24.0

Overview

Algorithms are the basis of all of computer science. It is no surprise when they frequently reappear in new contexts. All of the ideas are the same, but the appearance varies. This chapter and the next explore more generalized algorithms, represented in a more traditional form. Some hypermedia applications provide particularly powerful environments for creating algorithms or programming. These environments are especially nice as first programming languages. As you have seen, hypermedia allow the user to move from one location to any one of several “next” locations. Designers can also build in much more complex responses to user requestsactions ranging from “go to a new page” to make a sound to perform a complex arithmetic computation. The ease with which powerful and very visible algorithms can be added to hypermedia documents makes them among the most powerful and flexible end-user tools available today.

Some hypermedia applications such as *HyperCard*, *ToolBook*, and *Hot Java* provide very rich and powerful environments for writing algorithms. Using any of these environments, the programmer can duplicate any of the calculations previously performed in a spreadsheet. She can also create algorithms that navigate or move the user through hypermedia documents. Finally, she can create and draw pictures algorithmically. Much of this can be accomplished with very little additional training. This section makes no attempt to cover such programs in detail.¹ Rather, it restates the basic principles of algorithms described in earlier

1. This should be no surprisethe entire philosophy of the text has been to show the nature of the power and the tools for exploring that power. That philosophy continues here.

chapters, but in a new context. You will recognize most of the basic ideas. By the time you complete this chapter and the one following it, you should be able to write simple algorithms that generate interesting and useful results. This material also serves as a springboard for those students who are interested in learning more about computer science or who want to use more complicated algorithms in other contexts.

24.1

Formal Languages and Written Algorithms

Previous chapters used two separate, but closely related, techniques for representing algorithms: a series of cells in a spreadsheet, such as

	A	B
1	Input values	2
2		4
3		6
4	Their sum:	=SUM(B1..B3)
5	How many were there:	=COUNT(B1..B3)
6	Their average:	=B4/B5

and a generic written annotation, or *pseudocode*, used to describe algorithms independent of any particular application:

Count all of the selected items.

Add up the selected items.

Divide the sum by the number of items.

Both are representations for a single algorithm. They have identical meanings, but different appearances. In particular, the first is specifically tailored for spreadsheets, while the second is much more oriented to humans. Surprising as it may seem, the human-oriented representation is actually closer to that required by most traditional computer programming languages or applications.

Typically, the program designer specifies an algorithm as a series of steps, each written in a *formal language*. Every application has its

own *syntax*, or format for legal statements. In higher-level languages, this syntax takes the form of a series of written commands. Although the syntax of higher-level languages is different from that of spreadsheets, the process of creating algorithms from understanding the problem through proper testing and debugging is identical for any application. The following sections review the basic programming tools, restating them for a new environment.

24.1.1

Algorithms for Navigation

Navigation through a document is a process. Therefore, the response of an application to user requests for navigation should be describable as algorithms.

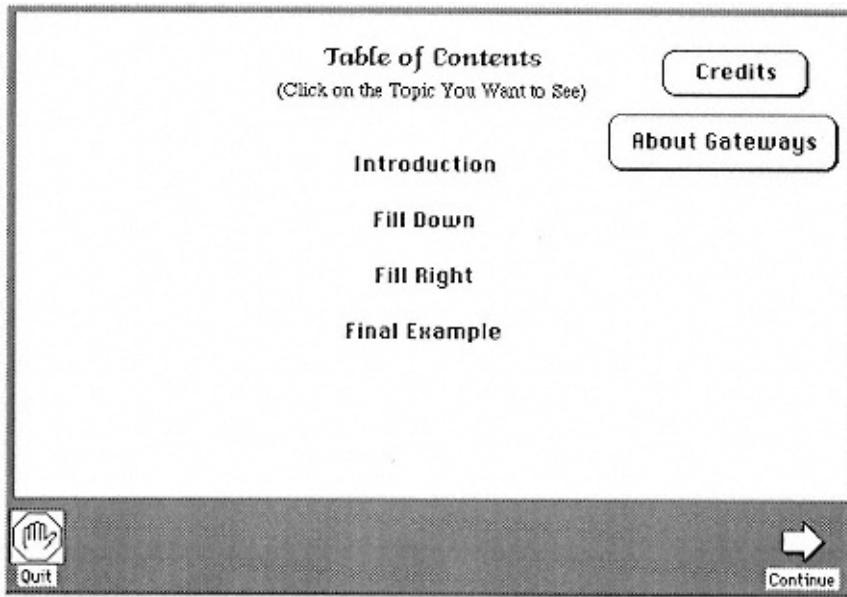


Figure 24.1
A Sample Card with Buttons

Figure 24.1 shows an example page from one of the *Gateway Labs*. The card includes several buttons allowing the user to select the next page to be displayed. One button takes the user to the introduction, another to the credits, and so on. A single algorithm describing responses to all of the needed actions could be written as in Code Segment 24.1. It is simply a series of conditionals, each describing the response to one specific button. The final algorithm is even easier to write, but it requires thinking about the organization of algorithms in a button-controlled environment.

*If the user selects button "Introduction"
then go to page "beginning of introduction".
If the user selects button "Fill down"
then go to page "beginning of fill down".
If the user selects button "About Gateways"
then go to page "beginning of About gateways".
If the user selects button "Continue"
then go to the next page.*

Code Segment 24.1

Algorithm for Responding to a Set of Buttons

24.2

New Algorithmic Representations

In traditional programming environments, the designer must specify two things: the individual or primitive steps and the exact order of those steps. This is true even for sequences of steps for which the order makes no difference (e.g., the sum of three numbers can be found by adding them up in any order). The algorithmic notation used throughout this text indicates the order of the steps by their relative position within an algorithm. Spreadsheets are a little less restrictive: the designer only needs to worry about the order when it actually makes a difference. In this spreadsheet segment:

	A	B
1	= $(3+4)$	= $(5+6)$
2		
3	=A1 *B1	

the value of A1 can be computed before or after B1 without impacting the final result (A3). At first, “button-pushing” environments seem to be unordered. Since a user can push a single button in only one order, the responder algorithm need only check to see if a button was pushed. At most one conditional will be true.

Traditional programming environments or languages, called *imperative* languages, focus on the actions and the order of those actions. The commands (note the implication of that term!) all have the form “do this, now do that.” The designer specifies everything exactly including the order of execution. In a relatively new paradigm called *object-oriented* programming (*oop*), statements are not thought of as imperatives that must be executed no matter what, but as *messages* or requests to change objects. Object-oriented programming focuses on the individual objects to be manipulated. In the case of hypermedia, this means (for now) the pages that are displayed along with the buttons and links that cause them to be displayed. A page is only displayed when the user requests it

(usually by pressing a button). On a request from the user, the application takes the user to the desired new page. A written algorithm need only specify how to respond to a button.

24.2.1

Objects and Messages

Hypermedia applications support a limited version of the object-oriented paradigm. For the moment, think of the buttons and the pages as *objects*. Think of actions as *messages* sent from one object (e.g., a button) to another (e.g., a page), and of algorithms as descriptions of how the application should respond to a given button. When an *event* occurs, the application executes the corresponding algorithm. In this case, whenever a user presses a button, a page is displayed. Rather than listing all of the conditionals together as in Code Segment 24.1, the programmer attaches each conditional to a separate button. That is, the algorithm

focuses on the individual button or object. The form of the conditionals for each button in the example is the same:

*If this button is pressed
then do some action.*

24.2.2

Examining an Existing Algorithm

You have already created algorithms that move a user to another page. That task was automated as part of the link and button design process. In effect, compiled hypermedia applications have built-in macros for creating the mini-program: “go to another page.” In fact, by examining the button, you can find the program it created. For example, the program:²

```
on mouseUp
    go to card Introduction Start
end mouseUp
```

is attached to the button `Introduction` of Figure 24.1.3

Consider each line of the program individually. The first line is roughly equivalent to the conditional:

*If the button is pressed
then . . .*

The term `mouseUp` indicates that this algorithm should be executed whenever the mouse button is released (i.e., it is allowed to go “up”).⁴ The last line of the program

```
end mouseUp
```

is a *delimiter*, marking the end of the algorithm to be executed when the mouse button goes up. The algorithm itself is really a single step:

```
go to card Introduction Start
```

2. All code examples used in this chapter and the next are as generic as

possible. Each is actually a legal segment of *HyperTalk* (the algorithmic language of *HyperCard*) code, but essentially the same code would apply for any other hypermedia application (e.g., *ToolBook* or *Hot Java*). The selection of *HyperTalk* represents a compromise. I did not want to endorse any specific language. But a real language does help illustrate the differences between real languages and the pseudocode used throughout this text. *HyperTalk* is a real language, and as real languages go it is relatively easy to understand complete examples. Also it is both reasonably available within Macintosh environments and very similar to the language of the *ToolBook* environment available in Windows environments. Placing full examples of multiple language in the text would be both long and confusing. The lab units contain examples specific to other appropriate languages.

3. To see how to find this program in your application, see Laboratory Unit 17.
4. Some new users find `mouseUp` confusing. Why isn't it `mouseDown`? That is, why not take the action as soon as the user presses the button. `MouseUp` assumes that the final user-action is the correct one. Waiting until the button is released allows the user to correct a mistake: if you press the wrong button, simply move the mouse off the button before releasing it.

meaning select (and make visible) the page5 called Introduction Start. These small programs attached to buttons are usually called *handlers* or *responders* since they handle messages to the button or respond to the button being pushed.

24.2.3

Some Other Primitive Tools

The primitive steps of algorithms in a spreadsheet are usually assignment statements: commands to place a value in a cell or *container*. So far, the primitive actions of hypermedia are all of the form:

Go to a new page.

Hypermedia languages allow other actions, including the familiar assignments as well as some new ones.

Sounds.

The designer can add additional explicit steps to the algorithm. For example, two commands to make the system produce sounds are:

Beep
Play "boing"

The first makes the default beeping sound for your machine; the second plays the named sound. These actions can be added to any algorithm. Thus, the three-step algorithm

```
on mouseUp
    beep
    go to card Introduction Start
    beep
end mouseUp
```

will beep twice, once before and once after going to the new card. Perhaps this would alert the user that the card has changed.

Visual Effects.

The user can also specify that visual changes occur as part of the algorithm. For example, the change from one card to the next may be more pleasing if one card faded into the next. The algorithm

```
on mouseUp
    visual effect dissolve
    go to card Introduction Start
end mouseUp
```

5. The individual pages of a HyperCard, SuperCard and some other hypermedia documents are called *cards*. Hence, the command is `go to cardName`, rather than `go to pageName`.

does exactly that: the first card starts to fade away. As it does so, the new one appears. Notice that the order of the algorithm is essential: the first picture must start to dissolve before the second appears.

Exercises

24.1 Find a button that you created yourself. Find the program attached to it and see what the application created for you. Describe any differences between that program and the example above.

24.2 Experiment with your program by changing the options (e.g., the visual effect) you selected when you defined the button. Look at the algorithm, and test it. Then change it from the button design dialog, look at it, and test it now. Finally, try to alter the algorithm directly: remove the visual effect, or change it to another effect (`zoom close` is one legal name). Look in the dialog for more names of the effects.

24.2.4

Old Structures in a New Environment

The algorithmic concepts and constructs discussed in earlier chapters are independent of their final representation:

individual actions or steps:

assignment (storing a value)

control structures:

ordered execution
iteration
conditionals

abstraction:

grouping of commands
macros
procedures

Each has a useful representation in hypermedia. The following sections address the algorithmic representations of the more important commands and structures. The most important thing for the reader to remember is:

You already know most of what you need.

The basics of algorithm design are the same. Only the syntactic details change.

24.3

Assignment Statements

Assignment may be the most universal of commands. Any application that allows the user to calculate values has some form of assignment. In a spreadsheet, every

basic action has a single form (see Section 9.4), the *assignment* statement:

Place a value in this location

typically in the form:

= *value*

The value may be a constant or it may be computed as part of the assignment. In either case, the command places a value in the specified storage location or *container*. In hypermedia tools, the assignment command has the general form:

Put *value* into *location*

For example,

Put 5 into myLocation

places the value 5 into a container called myLocation. Each function in a spreadsheet was written within a single cell. The command placed a value into that specific cell. Algorithmic languages allow greater flexibility: a single algorithm may store values in many different locations. So it must have a way of referring to or pointing to specific locations. Spreadsheet cells referred to other cells by their row and column. General algorithmic tools do not organize all values as a matrix of cells. Instead they give each location a name, such as myLocation, rather than the positional descriptor (row and column) used in spreadsheets. Any statement can then refer to the location by its name.

Once a container has a value, references to that container are equivalent to references to the value it contains. Thus, if the above statement were followed by:

Put 1 + myLocation into anotherLocation

it would place 6 into anotherLocation. These two lines together are equivalent to the spreadsheet segment:

	A
1	=5
2	=1+A1

For the moment, you need not worry about the nature of these storage locations. Just think of containers as cells in a spreadsheet, but identified by name rather than by location (row and column).

24.3.1

Expressions

Arithmetic (and other) expressions seem to appear in almost every problem. Fortunately, their representation in almost any language is identical. Consider the

spreadsheet expression:

$$= (3+5+7)/3$$

to calculate the average of three numbers. The equivalent expression is almost identical in any other language. For example,

`put (3 + 5 + 7)/3 into Average`

places the result in a location called `Average`. The rules for constructing the arithmetic expressions themselves are identical to the rules used in spreadsheets. (See Section 11.2, Arithmetic Expressions as Algorithms.)

Self-Reference.

One new twist made possible by algorithmic languages is the use of *self-reference* to define an assignment value. Consider the two code segments:

`Put 5 into myLocation
Put (1 + myLocation) into myLocation`

and

`Put 5 into myLocation
Add one to myLocation`

Both can be paraphrased as

*Put 5 into myLocation
Add one to that same location*

The value stored in `myLocation` changes twice. Spreadsheets do not support multiple changes of value partly because they must always recalculate a cell value each time any part of its defining cells changes (see Section 12.3, When Algorithms Fail). In each of the two examples, the second defining line refers to the cell being defined. So far, self-reference provides no additional capabilities—the same result can always be obtained using intermediate storage locations. But when combined with control structures in the next

chapter, self-reference becomes a very powerful tool.

Nonnumeric Expressions.

Hypermedia languages are designed for more general environments than are spreadsheets. Therefore, they usually have much easier-to-use tools for manipulating nonnumeric information. For example, concatenation (see Section 11.2.3, Nonarithmetic Expressions) may be represented by a binary or infix operator such as “`&`” rather than the more cumbersome `concat` function. The statement:

```
Put "Hello" & "John." into greeting
```

puts “Hello John.” into the container greeting, as does

```
Put "Hello" into opening  
Put opening & "John." into greeting
```

The self-reference capability allows a new way to construct a greeting:

```
Put "Hello" into greeting  
Put "John." after greeting
```

The second statement does not destroy the contents of greeting, but instead appends the string “John.” after whatever is already there. This technique is very useful for constructing “personalized” messages or other English-like output that depends on information obtained while the program is running.

24.3.2

Visible and Invisible Storage Locations

In a spreadsheet, all storage locations are visible. Hypermedia designs can have any number of storage locations, which can be either *visible* or *invisible*. What is the point of having good is an invisible location? While there is no absolute need for invisible locations, they are useful for hiding irrelevant information from the reader. For example, the user may not need intermediate values, and displaying such values would clutter the document and confuse the reader.

Visible Locations.

Algorithmically, visible and invisible locations are identical. Figure 11.1 shows how to calculate the total cost of an outfit. Figure 11.1. The same steps can be written as

```
Put 4 into ShirtPrice  
Put 5 into PantsPrice  
Put ShirtPrice + PantsPrice into OutfitPrice  
Put 3 into NumberPurchased  
Put (OutfitPrice * NumberPurchased) into TotalCost
```

This example could have been accomplished in fewer statements, for example, the third and fifth lines could be replaced by the single line:

```
Put (ShirtPrice + PantsPrice) * NumberPurchased into TotalCost
```

However, the intermediate value, `OutfitPrice`, serves the same cell C5 does in spreadsheets: it helps the designer detect and correct Invisible Locations.

Oddly enough, it is easier to create invisible locations, or *variables*, locations, or *fields*. *Variable* is the traditional name used for storage higher-level languages. The term is similar to, but not quite identical *variable*, used in algebra. No special action at all is needed to create Just pick a name and use it in a statement. The application will

	A	B	C
1	Prices		
2		shirt	\$4.00
3		pants	\$5.00
4	Costs		
5		outfit	=C1+C2
6	Number purchased		3
7			
8		total	=C5*C6

Figure 24.2
Cells Dependent on Other Cells

create and remember the location. The lab manual (Laboratory Unit 17) describes the exact syntax for requesting visible fields in your application.

Use invisible locations when the intermediate value is not important for the final user, or when it would clutter the screen.

24.3.3 *Scope*

An object is known only within the context in which it is declared, called its *scope*. In a spreadsheet, every cell had the same scope: it is known everywhere in the spreadsheet. Generally, invisible locations are created within an algorithm; therefore, they are known only within that algorithm; thus, if a value must be visible to another algorithm, it must be declared outside of the algorithm. Often, that means it will be a visible storage location. For now, assume that the scope of a given storage cell is restricted to a given procedure (or the set of procedures for a single object, such as those that define a single button). Keeping track of scope is a major reason for the popularity of object-oriented languages.

Exercise

24.3 Create a card that duplicates the spreadsheet segment in Figure 24.2. Place the algorithm in the script for a button. Create all necessary storage locations. Clearly all user-input

fields (e.g., the price of a shirt) must be visible, as must the final answer. The intermediate storage locations (e.g., cost of all the shirts) can be either visible or invisible.

24.4

Algorithms and Visualization

The basic building blocks of hypermedia languages include actions that draw pictures. This means that a designer can write a program that algorithmically

draws any picture she can create with a draw or paint tool. In addition, algorithms can draw a picture based on information not available at the time the algorithm is written.

24.4.1

Algorithms that Draw Pictures

Many designers would like to draw pictures based on data provided by the users. Spreadsheets provided a limited version of this capability in the form of charts or graphs (see *Chapter 20: Visualization Revisited*). Higher-level languages provide a generalization of this capability. In general if you can describe the picture, you can write an algorithm to draw it. In particular, programs can imitate any steps used in painting tools (see Section 7.3.1, Paint Tools and Laboratory Unit 17).

First recall the steps you needed to create an object:

Select a tool for drawing the object.

Draw the object by dragging the mouse across the desired location.

The corresponding commands, `choose` and `drag`, provide this basic drawing capabilities for algorithms. For example, this short program will draw a rectangle:

```
choose the rectangle tool
drag from 100,100 to 200,300
```

24.4.2

Coordinate Systems

Section 20.2.4 provided a quick review of the Cartesian coordinate system. The numeric parameters in the `drag` command correspond to a very similar (but not quite identical) coordinate system. Each pair of numbers represents a point on the window in which you are drawing; one represents the horizontal position and the other, the vertical position of the mouse.⁶ Each describes the distance

between the point and the corner of the drawing window. The horizontal position is described in terms of the distance from the left edge of the window: in this case, the starting position is 100 pixels from the left edge; the ending position is 200 pixels from that same edge.⁷ The vertical positions are measured in terms of the distance from the top of the window. Thus, the mouse moves vertically 200 pixels, from row 100 to row 300. A typical computer screen is between 500 and 800 pixels wide and 300 and 600 pixels high. Alternatively put, one inch typically equals about 75 pixels. Any point in the entire window can be described in terms of its horizontal and vertical position. The designer can specify the location of the object by the position of its first corner and its size by the relative location of the second corner.

6. The exact details vary. For example, all Macintosh applications represent the horizontal coordinate first and the vertical second.
7. Notice that according to the terminology of Section 16.3 these are relative positions.

24.5

Programming Revisited

By combining the actions needed for several shapes into a single algorithm, the designer can describe any shape. For example, the code:

```
on mouseUp
    put 100 into height      all edges will be 100
    put 100 into width       pixels high
    put 35 into offset       all edges will be 100
                            pixels wide
                            the front is offset 35
                            pixels

    choose rectangle tool
    drag from 0,0 to         create one rectangle
    width, height
                            and another

    drag from 0 + offset, 0 + offset to width +
    offset, height + offset

    choose line tool          lines for edges
    drag from 0,0 to          connect top left
    offset, offset            corners

    drag from width, 0 to width + offset,
    offset and top right
    drag from 0, height to offset, height +
    offset bottom left
    drag from height, width to height + offset,
    width + offset bottom right

end mouseUp
```

will generate the three-dimensional cube in Figure 24.3 by drawing the front and back surfaces (two rectangles) and then connecting each of the four pairs of corresponding vertices. The variables `height`, `width`, and `offset` help the programmer guarantee that all edges will be the same length. Use of the arithmetic expressions such as `height+offset` saves the programmer the trouble of adding two

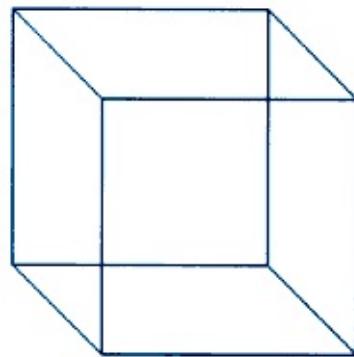


Figure 24.3
The Cube Described by
an Algorithm

values together. Notice that the lines beginning with “- -” are comments providing documentation.

24.5.1

Other Capabilities

In applications that allow users to manipulate objects through either a gui interface or a user-written program, it is usually possible to perform any action with either method. Programming has a number of advantages over gui interaction:

Sequences of actions will occur faster.

Images can change before the user’s eyes. A program can draw a picture, and then erase or move part of it, creating a sort of animation.

The program can communicate with the user. For example, compare a program beeping to attract the user’s attention with the user’s requesting the application to beep.

A program can lead the user through a series of pages.

It is usually easier to draw a picture very accurately with a program, especially when attempting to replicate an existing picture.

For these reasons, many users will eventually find a need to add programming to their essential list of skills.

Exercises

24.4 Repeat the algorithms you created for Exercises 11.1, 11.2, 11.4, and 11.7. This time use a hypermedia environment. Instead of spreadsheet cells, use variables or fields.

24.5 Repeat the algorithms you created for Exercises 11.11, 11.12, 11.15, and 11.18. This time use a hypermedia environment. Instead of spreadsheet cells, use variables or fields.

- 24.6 Write code to draw a circle inscribed within a square.
- 24.7 Write code that creates a blinking picture. Draw the picture, erase it, draw it again, and so on. If it blinks too rapidly, try using a pause or wait command between each blink.
- 24.8 Write code that writes your name in a window one character at a time.
- 24.9 Write code that draws a picture of a simple house.
- 24.10 See the lab manual for more examples.

24.5.2

Old Techniques in a New Environment

Almost all programming techniques you learned for use in spreadsheet environments are equally valid in a hypermedia document. Perhaps they are even more relevant because documents and programs are likely to be larger.

Top Down Design.

Design your code top-down. Start with the general ideas and then fill in the details (see Section 13.3.1).

Plan First-Then Code.

Always plan your work first. Think through the desired results. Think through, and sketch out the process in pseudocode. After you have an outline for a solution, code it as a formal algorithm. Notice that you can build and test your the algorithm bottom-up. Test each procedure and function as you design it. That way you will be much better able to predict the locations of errors (see Section 13.3.2).

24.5.3

Testing and Debugging

Never assume that your code is correct. Always test your programs. Create test data that tests all possible situations (see Section 13.6).

24.5.4

Build Incrementally

A technique not discussed in the context of spreadsheets is *incremental construction*. Hypermedia and oop documents lend themselves particularly well to this technique. The basic idea is to build and test the program in segments. Each small piece of the entire program can be addressed individually. At first this approach seems to contradict the principle of top-down design. The watchwords should be:

*Design top-down,
then build bottom-up.*

Consider the following example. Suppose you realized that you would need the procedure `badSituation`. Top-down design dictates that you note the need for the procedure. Design can wait. But with incremental design, you could build and test the procedure

even though you did not know exactly where or how often you would need it. Why is incremental design useful in this context? Because it is possible to isolate a procedure completely while you test it. For example, build the procedure and build a little test program:

```
on mouseUp
    badSituation
end mouseUp
```

If this isolated procedure does not work, you know the error must be within the one specific step. There is no point looking elsewhere for the error. Once you are confident the procedure works, you can use it wherever you need it. If there is a problem then, you know the problem must be elsewhere.

24.5.5

Documentation

Documentation is at least as important in algorithmic languages as it is in spreadsheets, perhaps even more so. There are more structures and more

variations. Programs may be even longer. The general guidelines are the same (Section 13.5).

Variables.

Maintain a clear list of all variables and constants showing the assumed type and the role in the program. One of the advantages of a higher-level language over a spreadsheet is that storage locations can have names rather than numbers. Pick names that help the programmer remember the role of the storage location. Pay attention to the declarations as you write the program body.

White Space.

Use blank lines to break a program into logical segments (much like paragraphs). In the cube example, white space separates the variable assignments from the rectangle surfaces and the connecting lines. Liberal use of white space improves readability. In general, include blank lines to separate:

logically separate portions,

data manipulation segments from variable description, and

blocks of comments from blocks of code.

Output.

Provide output or display statements so that you can follow these values as the program executes.

24.6

References

More than for any of the previous tools, hypermedia users need reference material. The following books or references may help the students who want to go further.

For Hypercard

Beekman, George *HyperCard 2.3 in a Hurry*. Belmont, CA: Wadsworth, 1996.

Goodman, D. *The Complete HyperCard Handbook*. New York: Bantam Books, 1976.

Shell, B. *Running HyperCard with HyperTalk*. Portland, OR, MIS Press, 1988.

For Toolbook

Hall, Tom L. *Utilizing Multimedia ToolBook 3.0*. Danvers, MA: Boyd Fraser Publishing, 1995.

Hustedde, S. *Developing with Asymetrix ToolBook, Applied Programming Theory*. Belmont, CA: Wadsworth, 1996.

Holtz, Matthew. *The Multimedia Workshop: ToolBook 3.0*, Belmont, CA: Wadsworth, 1995.

24.7

Summary

The concept of algorithms from earlier chapters reappears throughout computer science. In particular, the rules and guidelines found in earlier chapters apply

equally well to hypermedia applications. In particular, you can create sequences of actions composed of page navigations, sounds, assignments, and drawing. Algorithmic languages must have a formal syntax capable of describing both a set of actions and the objects those actions manipulate. Although you have seen the basic tools common to all algorithms, this chapter in no way presents all of the possibilities. For those, you should consult one of the references above, or take a more advanced computer science course.

Some Built-in Messages

mouseUp MouseEnterpause

mouseDownmouseLeavewait

Actions you Should be Able to Perform

Draw any geometric shape. Erase a region.

Color or fill in a region. Add text to a window.

Select a region. Close the window.

Drag a region to a new location. Quit the application.

View several pages in succession. Make noises.

Hide an object and make it reappear. Any action from any menu.

Important Ideas

algorithm assignment object-oriented

white documentation program
space

higher-level language

25

Control Structures Revisited

The head learns new things, but the heart forever more practices old experiences.

HENRY WARD BEECHER

Pointers

*Gateway Algorithms in
Lab: Hypermedia*

*Lab Unit 18: Algorithmic Control
Manual: Structures*

25.0

Overview

Chapter 24 described many of the basic building blocks for hypermedia environments. Each of these building blocks was analogous to one from the spreadsheet environment. Control structures are similarly represented: all higher-level languages, including those of hypermedia environments, also support the control structures and procedural abstraction tools analogous to those described in *Chapter 15: Iteration: Replicated Structures* through *Chapter 19: Abstraction, Abbreviation, and Macros*. As with their basic operations, higher-level languages represent each command as an imperative statement. The form of each varies, but the interpretation is the same. This chapter provides a very brief introduction to these structures.

25.1

Iteration

Recall that iteration simply means doing repeatedly (for a review,

see *Chapter 15: Iteration: Replicated Structures*). This simple construct is a very powerful tool in a general programming language. A spreadsheet designer creates an iterated structure by taking advantage of the matrix structure of the spreadsheet. She can replicate a command by filling down (or right) across multiple cells. Since higher-level languages do not assume a spreadsheet's table organization, they

require a slightly different representation. The commands actually look almost identical to the algorithmic notation used to describe general algorithms:

*Repeat n times
action.*

For example, the procedure

```
Repeat 10 times
    beep
end repeat
```

generates the beep sound 10 times.

In general, any action can be repeated any number of times.

Suppose you wanted to calculate the value of 65. One algorithm is:

*Put 1 into Temporary.1
Repeat 5 times:
 Multiply Temporary by 6.
 Put Temporary into Result.*

As a formal program this becomes:

```
Put 1 into Temporary
Repeat 5 times
    Multiply Temporary by 6
end repeat
Put Temporary into Result
```

Abstract Groups.

The line,

```
end repeat
```

in the above algorithm is a *delimiter* similar to `end mouseUp`. It carries the same logical meaning as does indentation: it marks the end of the scope of the control structure in this case the iteration. Notice that the two methods for showing scope indentation and

explicit end-marks are equivalent. Each shows which statements the structure applies to, in this case which should be repeated. Although most humans find that indentation is clearer, most languages use the explicit end-marks for historical reasons (early language interpreters could not recognize an indented line). In fact many formal languages require the formal end-mark even when there is only a single step as in this example. Although we talk of the step to be repeated as a single action, it may in fact have several substeps. Such groups are always executed as if they were a single statement: all or nothing. If one statement in the group is executed, they are all executed.

1. Note that any number raised to the zero power is 1. Starting with 1 in `Temporary`, rather than 6, assures that the algorithm will work even if the desired power were zero. It also makes errors less likely because the number of repetitions is the same as the desired power.

Consider a slightly more complicated example: the algorithm for calculating the factorial of 5 ($5 * 4 * 3 * 2 * 1$) originally shown in Section 16.2 and reproduced here as Figure 25.1. The algorithmic equivalent is:

```

Put 1 into Temporary
Put 1 into Counter
Repeat 5 times
    Multiply Temporary by Counter
    Add 1 to Counter
End repeat
Put Temporary into Result2

```

At each pass through the repeat loop, the algorithm calculates a value equivalent to a new cell in the spreadsheet: A2 through A6. This example brings out the real advantage of self-reference. The single container `Temporary` successively receives the new values: 1, 2, 6, 24, and 120. At each step, the assignment refers to `Temporary`, makes a calculation, and places the value back in the same location. The spreadsheet required a separate cell for each step. While `Result` should be visible, there was no need to retain or display all of the intermediate values.

With an algorithmic language, the designer need not know how many times the segment will be repeated. If the container `howMany` contains any number, the revised program

```

Put 1 into Temporary
Put 1 into Counter
Repeat howMany times
    Multiply Temporary by Counter
    Add 1 to Counter
End repeat
Put temporary into Result

```

	A	B
1	Number	Factorial
2	1	=A2
3	=1+A2	=A3*B2
4	=1+A3	=A4*B3
5	=1+A4	=A5*B4
6	=1+A5	=A6*B5

	A	B
1	Number	Factorial
2	1	1
3	2	2
4	3	6
5	4	24
6	5	120

Figure 25.1
Calculating a Factorial

2. Actually this could have been done without `Temporary`. The intermediate value simply made the algorithm clearer.

will calculate the factorial of that number. In a spreadsheet, the designer must `Fill down` the appropriate number of cells. That allowed two choices. Either the designer knew in advance the number for which the factorial would be needed, or the designer used `Fill down` to create a table of values, forcing the user to find the needed value from the list of values. With the higher-level language representation, the program can produce a single answer corresponding to the appropriate number of iterations.

Exercises

25.1 Duplicate the power algorithm without using the variable `Temporary`.

25.2 Create the factorial algorithm and test it with various values.

25.2 Conditionals

The format for conditionals (see *Chapter 17: Conditional Actions*) is even more similar to that found in spreadsheets. For example, an algorithm that tests a value and warns the user that a number is too large for its intended purpose might look like:

```
if Average < 1000
    then put Average into Result
    else beep
```

This is almost exactly the same as the notation used for algorithms throughout this text:

*If the average < 1000
then put the average into Result
otherwise make a beep*

In general, the relation between pseudocode, algorithmic syntax, and spreadsheet syntax can be described by reversing Figure 17.1

as in Figure 25.2. The needed syntactic “glue” is provided by the words `if`, `then`, and `else`. The general form is simply:

```
if test
    then yes action
        else no action
```

Where *test* is any Boolean comparison, and the two *actions* can be any actions describable in the language. As with the `repeat`, two or more actions can be combined in the `else` clause by using an `end if` to delimit the end of the entire block.

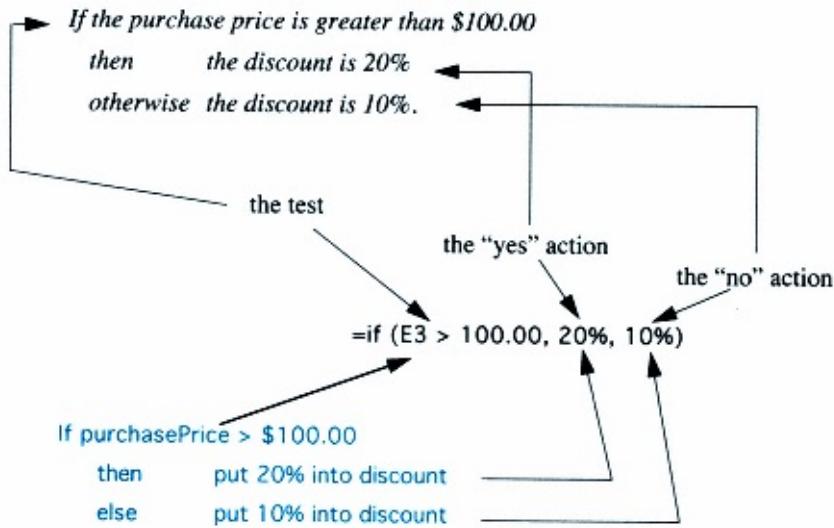


Figure 25.2
Representation of a Conditional

```

if average < 1000
    then put Average into Result
else
    beep
    beep
end if

```

The `else` clause, if present, will serve the additional role of delimiting the `then` clause. The word `else` serves double duty: it marks the beginning of the `else` clause and forces the end of the `then` clause. Most languages allow you to omit the `otherwise` or `else` clause if it contains no actions:

```

if a>b
    then put temp + 1 into temp
end if

```

Top-Level Conditionals.

In hypermedia systems (and to some extent, object-oriented systems in general), many of the conditionals seem to “disappear into the woodwork.” In particular, any conditional of the form

If this button is pushed

then perform an action

is not explicitly used. Since the algorithm is associated with a button, it is not executed unless the button is actually pushed. So if the algorithm is executed, the

question “*If this button is pushed*” provides no additional information; it is always true. The single-line algorithm:

Perform an action

accomplishes the same result as the above conditional. Of course the single-line algorithm has its own delimiters:

on condition
perform action
end condition.

Finally, the algorithm composed entirely of conditionals in Code Segment 24.1 never gets written explicitly. Each conditional is buried within a single button script. Each of those scripts is a condition which must be true given that the corresponding button is pressed. Thus each of those handlers has the identical and simple form:

```
goto page pageName
```

The similarity and simplicity of commands make many programs almost trivial to write.

25.3 Abstraction

The creation of abstract objects greatly extends the capabilities of any program. They enable the programmer to think of more general problems, reuse existing code, and isolate errors. The techniques in a higher-level language are essentially the same as those described in *Chapter 19: Abstraction, Abbreviation, and Macros*.

25.3.1 *Documentation*

Documentation is an essential tool, enabling programmers to specify what they are attempting to accomplish. Documentation allows the designer to write general or abstract statements about the

goals of a section of code. Think of documentation as annotation: comments that explain what is happening. Often English (or even English-like) statements are much easier to understand than the specific details of the code. Document code by segments (as described in Section 13.5). Algorithmic languages use explicit *comment statements*. The programmer must distinguish between actual computer commands and comments intended for human readers. Typically, a comment is delimited with a standard delimiter, such as

```
-- this is a comment marked by "--"  
; this is a comment marked by ";"
```

In this example, anything written to the right of the marker will be a comment.

Thus, a comment can be part of a line or the entire line depending on the location of the marker. Some languages allow the programmer to delimit both the beginning and end of the comment; in others the comment includes the entire right-hand end of the line. As a general commenting guideline, start by providing a comment describing any abstract group of statements that work together to accomplish a single result: a procedure, or even a smaller block such as an iterated or conditional group.

25.3.2

Scope and Delimiters

The ending delimiters `end if`, `end repeat`, and `end mouseUp` all serve the same purpose: to mark the end of a group of commands that go together. This grouping capability is essential for creating more complex structures. Any set of statements can be grouped together. Once grouped they act as a single statement. This means that it is possible to combine groups into larger groups. For example, the body of a repeated group can itself contain a repeated group. Suppose that you wanted to sound a beep once, then twice, then three, and finally four times (a total of ten beeps). One algorithm for this is:

```
Put 1 in Count
Repeat 4 times
    repeat Count times
        beep
    add 1 to Count.
```

The "statement" within the outside *repeat block* is also a *repeat*. The variable *count* specifies the number of times the inner loop is repeated. This text indicates the grouping of commands by indenting commands that go together. The outer loop in this example would execute four times. During each of those four repetitions, the inner loop would execute *count* times. Thus on the first pass of the outer loop, the inner loop would only execute once. But on the next pass, *count* will be 2

the inner loop will be executed twice . . . then three times, and finally four times. Most formal languages mark groups using the equivalent delimiters. Thus, the above is equivalent to the grouping:

```
put 1 into Count
repeat 4 times
  repeat Count times
    beep
  end repeat           -- (second or inner repeat)
  add 1 to Count
end repeat             -- (first or outer repeat)
```

A delimited group of statements cannot be broken. They behave as a single statement: all (or none) of the statements are executed. Every `end` always corresponds to the most recent unmatched opening delimiter (e.g., `repeat`).

25.4

Procedural Abstraction

Suppose a designer finds that an action appears repeatedly throughout a program. For example, perhaps she feels that a single beep used earlier does not get the user's attention, and that three beeps could do a much better job. A block of code something like:

```
if some particular condition
  then beep
    beep
    beep
```

accomplishes this goal. In a formal language this may look like:

```
if average > 1000
  then beep
    beep
    beep
end if
```

Many programmers find this notation inconvenient if they often need multiple beeps. First, it requires four or five lines in place of the single line needed for a simple beep. It is also easy to forget: “Was I using two or three beeps?” If the designer changed her mind at a later date and wanted two rather than three beeps, she would have to go through the entire document to find and change every instance of the beeps. Finally, it would be nice to have a term that indicated the reason for the three beeps something like `get-users-attention`.

25.4.1

Basic Procedures

These problems should sound familiar. They are exactly the reasons given in *Chapter 19: Abstraction, Abbreviation, and Macros* for using procedures or macros. Most languages provide a similar tool. Programs to this point have all been short algorithms that are

executed whenever a specific event the mouse button is released occurs. Users can create procedures that are executed based on other events. For example, the procedure

```
on mouseEnter  
    beep  
end mouseEnter
```

would be invoked whenever the mouse entered the button (it would not wait for the user to press the button). `MouseEnter` and `mouseUp` are the names of *messages* which are sent to the button. When a message is received, the corresponding procedure is executed. Many similar messages are predefined. For example, `mouseDown` is like `mouseUp`, but occurs when the button is first pressed.

Messages are even more powerful because any user can define them. The user above who wanted to signal three beeps whenever a bad situation occurs can define `badSituation` to be a message, and define a response to that message:

```
on badSituation
  beep
  beep
  beep
end badSituation
```

Any procedure can send a message to any object within the scope of the procedure:

```
if average > 1000
  then badSituation
```

`BadSituation` is identical in concept to a procedure or macro. In this case one user-defined procedure calls or sends a message to another procedure. The second procedure responds or executes when it receives a message. In every place that requires three beeps (as in the example), a procedure can send the message to `badSituation`. The handler has all of the advantages of a macro:

- reduced number of lines of code,
- easy-to-correct errors,
- consistency,
- ease of change, and
- names that make the current goal clear.

Scope.

As with variables, procedures are only known within a limited region or scope. In general, you will be safe if you use procedures only with the objects (a given button or a given card or page) for

which they were defined. See any of the references cited at the end of Chapter 24 for a more general discussion of scope.

25.4.2

Communications to Procedures

Suppose a designer wanted a varying number of beeps, depending on how bad the situation was; sometimes she wanted three beeps, sometimes four, and sometimes some other value. One possible solution would be to create three procedures, perhaps `badSituation`, `veryBadSituation`, and `extremelyBadSituation`. The only difference between these procedures might be the number of beeps produced. It seems like there must be a way to combine the three similar procedures. The flexibility of the `repeat` statement provides a clue to the best approach. A procedure containing the segment:

```
repeat howMany times
    beep
end repeat
```

beeps a different number of times depending on the current value of `howMany`.

How does `howMany` get a value? One possible solution is obvious: places the value into the container `howMany` before calling the proc forces the end-user to know the name of the location and to use two commands: one to place the value into `howMany` and a second to call `badSituation`. It also implies that the calling program must be in the scope of `howMany`. None of this is very convenient.

Parameters.

The needed tool is called a *parameter*. In the following example, the variable `howMany` serves this purpose.

```
procedure badSituation (howMany)
  repeat howMany times
    beep
```

The parameter name `howMany` is called the *formal parameter*. It marks the location associated with the procedure. The value is said to be *passed* to the procedure: when another program calls `badSituation`, it also passes a value to the procedure to use as `howMany`. In a formal language, the procedure n

```
on badSituation howMany
  repeat howMany times
    beep
  end repeat
end badSituation
```

The parameter `howMany` behaves much like a variable: it will hold whatever value is given to it. It is given a value when the user (or another procedure) calls `badSituation` with the message:

```
badSituation 5
```

The value 5 is placed in the parameter `howMany` before the procedure `badSituation` actually starts to execute. The passed value, 5, is called the *actual parameter*. The formal and actual parameters are associated with each other by their positions: each appears immediately after the word `badSituation` in the definition and 5 in the calling statement. Once the procedure has been defined,

starts, it treats howMany like an other container. In this case it holds referenced by the repeat command. For example a code segment

```
If value > 1000
    then badSituation 5
    else if value > 100
        then badSituation 3
        else some action
    end if      -- end of inner conditional (value > 100)
end if          -- end of outer conditional (value > 1000)
```

contains two calls to badSituation, with the actual parameters

Parameters in Built-in Procedures.

Of course, parameters are exactly the tool you have used previously in conjunction with the so-called built-in functions in spreadsheet applications.

Average (3, 5, 7)

is a call to the procedure Average with actual arguments of 3, 5, and 7.

Exercises

25.3 Write a procedure that displays a 100 by 100 pixel square with parameters telling it where to place the upper left-hand corner.

25.4 Write a procedure to draw a rectangle of any height.

25.5 Write a procedure to draw and erase a rectangle any number of times.

25.6 Write a procedure to draw any number of circles so the edges just touch.

25.7 Write a procedure to visit each of the next n cards (for any n)

25.4.3

Communication from Procedures

An earlier example calculated 65 . Examination of this procedure reveals another problem. Programmers often want to know the results of the calculation. By themselves the results of calculating 65 are not too interesting. Perhaps you would like to build a procedure that could calculate ab for any base a and power b . Part of the solution to this problem is obvious: The user passes actual values for a and b to the procedure:

on Power a, b

```
Put 1 into Temporary
Repeat b times
    Multiply Temporary by a
    end repeat
    Put Temporary into Result
end mouseUp
```

But how does the user get at Result?

Returning Results.

The above solution works fine as far as it goes. Although parameters can resolve the problem of communicating information to the procedure, there is a second problem: How does the function communicate its final Result back to the user? The Power function can be rewritten using parameters and a *returned value*:

```

function Power a, b
    Put 1 into Temporary
    Repeat b times
        Multiply Temporary by a
    end repeat
    return Temporary
end power

```

Two syntactic differences in this procedure are immediately obvious: First, it started with the word `function` rather than `on`. A *function* is just a procedure that wants to return a value to its caller. Recall that each built-in function in a spreadsheet calculated (and returned) a value. That is, the function returned a value that could be displayed in the cell or used in a more complicated expression. The programmer could use a function call wherever a simple value was legal. A user-defined function works in exactly the same way. The second difference is the additional line

```
return Temporary
```

which says “the function has completed its work and it is time to return to the calling program, delivering the value of `Temporary`.” Any procedure can take advantage of the function by *calling* it with the appropriate parameters:

```
put Power (6, 5) into NewLocation
```

meaning: calculate 65 using the function `Power`, and place the returned value into a location called `newLocation`. In cases such as this with more than one parameter, the value of the first actual parameter is given to the first formal parameter (6 went into `Base`) and the value of the second actual parameter is given to the second formal parameter (5 went into `Expon`). The number of actual parameters must be the same as the number of formal parameters. Finally, notice that the names of the parameters make no logical difference, but the use of good mnemonic names improves the readability of the program:

```
function power Base, Expon
    Put 1 into Temporary
    Repeat Expon times
        Multiply Temporary by Base
    end repeat
    return Temporary
end power
```

Exercises

- 25.8 Write a function that calculates and returns the average of three numbers.
- 25.9 Write a function that, given the left and right ends of a cube, calculates its volume.
- 25.10 Write a function that concatenates three words into one long name.

25.5 Recursion

One of the most powerful computation techniques is recursion: defining a procedure in terms of a very similar procedure. Section 16.2 used recursive techniques for defining compound interest, exponentiation, factorial, and other problems. Recursion provides even bigger advantages in algorithmic languages. Consider an alternative definition of raising 2 to a power:

```
if number = 0
then 2number is 1
otherwise 2number is 2 × 2(number - 1)
```

or put just slightly differently:

```
if number = 0
then 2number is 1
otherwise 2number is 2 × power (2, number - 1)
```

This definition³ divides the possible cases into two groups: a trivial or base case (any number raised to the zero power is 1) and all other cases. Each of the other cases simply redefines the problem using a simpler (and presumably previously solved) problem; in this case, 2 to any power is defined in terms of 2 to a smaller power. In a spreadsheet this would have been represented as in Figure 25.3:

	A	B
1	power	2 ^{power}
2	0	1
3	1	2
4	2	4
5	3	8
6	4	16
7	5	32
8	6	64
9	7	128
10	8	256

	A	B
1	power	2 ^{power}
2		0
3	=1+A2	=2*B2
4	=1+A3	=2*B3
5	=1+A4	=2*B4
6	=1+A5	=2*B5
7	=1+A6	=2*B6
8	=1+A7	=2*B7
9	=1+A8	=2*B8
10	=1+A9	=2*B9

Figure 25.3

Each cell after the first is defined in terms of the cell above it. The

syntax in a higher-level language looks almost exactly as it did in the algorithm:

3. This example should look familiar as the well-rewarded peasant example in Section 16.2.

```

function power Expon
if Expon = 0
    then return 1
else return 2 * Power (Expon - 1)
end power

```

Exercises

25.11 Enter and test the above definition of power. Compare the result to the algorithm for the well-rewarded peasant problem (see Section 16.2).

25.12 Write a recursive version of the more general power function: allow any base and any exponent.

25.13 Write a recursive algorithm for calculating a factorial.

25.14 Write a recursive definition of the n th Fibonacci number (see Exercise 16.10).

25.15 For each recursive exercise in Chapter 16, redo the problem in an algorithmic language.

25.5.1

Recursion and Pictures

The techniques of recursion combine in very interesting ways with the algorithmic picture creation to create complex looking, but simple to understand pictures. Alternatively, such pictures are excellent tools for visualizing recursive works. Consider the following two-part algorithm, called `bullseye`

```

on mouseUp
    choose oval tool
    bullseye 100
end mouseUp

on bullseye size
    if size > 0 then

```

```
    drag from 200 - size, 200 - size to 200 + size
    bullseye size = 10
end if
end bullseye
```

The interesting section is the second part (the first part simply selects indicates that it is time to draw the bullseye). The second part contains definition of a bullseye, centered at location 200,200. The algorithm paraphrased as:

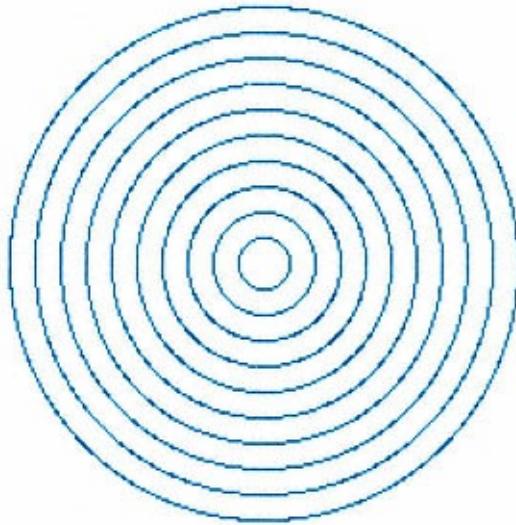


Figure 25.4
A Recursive Bullseye

*if the current size is not too small
draw a circle, and
draw another (but smaller) bullseye inside of the first.*

This creates the bullseye effect as in Figure 25.4. Notice how the bullseye algorithm fits the definition of recursion perfectly: the base case is certainly trivial: if the size is too small, it does nothing. The steps are guaranteed to end because at each iteration of the cell, it draws a smaller bullseye. Eventually the size will be small enough to stop the process.

Exercises

25.16 Enter the above definition of bullseye. Test it and then attempt several small modifications:

- a. Draw a rectangle rather than an oval.
- b. Draw the left edge as 0,0 rather than drawing the circles around the point 50,50.
- c. Make the oval very wide compared to its height.
- d. Draw two ovals in one procedure: one getting smaller in

the vertical dimension, the other getting smaller in the horizontal.

25.17 The bullseye recursion stops because each successive oval is smaller. Is it sufficient to just get smaller each time? If not, what further conditions must apply?

(table continued on next page)

(table continued from previous page)

25.18 Write a recursive function that displays the letters of the alphabet from “z” down to “a”.

25.6 Programming Revisited

Were it offered to my choice, I should have no objection to a repetition of the same life from its beginning, only asking the advantages authors have in a second edition to correct some faults in the first.

BENJAMIN FRANKLIN

25.6.1 *Structure Suggestions*

Structures.

Use only the standard control structures iteration, conditionals, and procedures, and document them carefully. These may be as simple as the value of an index after leaving a loop. Remember that a group of statements treated as a single statement cannot be broken. That, in turn, requires that no group can end part way through a group that it contains. Every `end` will always correspond to the most recent unmatched opening delimiter.

Scope.

Most languages place restrictions on the use of *global variables*. The primary reason for such restrictions is the prevention of undesirable side effects. If possible, use variables only within the procedure or objects in which they are defined.

Length.

Keep the program units (functions and procedures) short, perhaps no more than 10 to 15 lines, excluding the comments. Program bodies that are longer than can be seen on the screen at one time are

often too complex to understand easily. If your routines are consistently longer, they are too long.

25.6.2

The “Drawability” Test

With a well-written pseudocode algorithm, you should be able to draw arrows from each statement to any statement which might follow it immediately chronologically. The resulting collection of arrows should not form spaghetti. In fact, you should be able to do this with no overlapping arrows. If you cannot, your design is probably more complex than it needs to be. On a paper copy, draw an arrow from each statement to every statement that can follow it. There should be very few crossed lines, and few single arrows extending across long distances.

25.6.3

Documentation Guidelines

Procedures.

In writing the procedure, list any variables within a declaration block at the beginning of the procedure.

Control structure and data definitions should be as well-specified as the algorithm itself. In iteration, the pseudocode representing the body of a loop can be indented and separated from its control mechanism by blank lines:

Indentation.

Always indent loops, conditional structures, and procedures even if they are marked by `end` delimiters. Take control of the appearance of your structures by selecting a format that reflects your intent.

25.7

Summary

The iterative, conditional, and procedural control structures provide the basis for all higher-level languages. This chapter provides a very brief introduction to these structures. Those students who continue in computer science will frequently encounter these structures in new environments. Hopefully this small taste will encourage you to explore further.

Important Ideas

iteration	conditional	procedure
function	actual	formal
	parameter	parameter
pass	return	structure
delimiter		

26

Where to from Here?

Avarice in old age is foolish; for what can be more absurd than to increase our provisions for the road the nearer we approach to our journey's end.
CICERO

Pointers

*Lab Unit 19: Extending What You
Manual: Know to Another Gui
Environment*

26.0

Overview

As students near the end of any course they often ask some very reasonable questions:

Where to from here?

What should I expect next?

Can I apply this material in my other courses and in my future career?

What is the most important material I learned this semester?

Unfortunately, it would take a fortune teller to provide answers to some of these questions. However, we can address some of the more fundamental or important issues. This final chapter addresses three questions related to the material in this course and your future:

What will computing look like in the future?

Will I need to learn new software tools? If so, how do I do that?

What other computing and computer science courses should I take?

26.1

What will Computing Look Like in the Future?

Unfortunately, this question is very hard to answer in general. The history of computing is full of predictions that look ridiculous in hindsight far more by being too conservative than too optimistic. Many computer scientists once believed that no more than 100 computers would be needed for the computation

requirements of the entire nation. As late as the early 1980s, computer professionals debated whether or not personal computers would ever be common or popular tools. I do not intend to add any silly projections of my own about the future. But I will point out a few trends that we should expect to continue.

Machines will certainly continue to become faster and easier to use. Beyond that, very few things are certain about the future of computing. The nature of programs will change, as will the interfaces. More and more of the mundane aspects of tasks will be automated, and probably some of the not-so-mundane as well. But which ones will change first, and what applications will continue in nearly their present form are anyone's guess. Fortunately, you do not need to start over with each new application you use. Your background in problem solving will make it easier to adjust.

26.1.1

The Past as a Window on the Future

The past evolution of computers suggests some general directions for the computing of the future. Recent changes and trends suggest the sorts of changes we should expect in the near future.

Power.

For almost half a century, computing power has steadily improved. Machines have gotten faster and memory size has increased. Every year computers are faster than in the previous one. One rough rule of thumb known as *Moore's Law* suggests that memory capacity doubles every year. Much of the increase in speed has come from miniaturization (the smaller the components, the shorter the distances signals must travel and therefore the faster the product). While the exact rate of increase or even the areas of most significant improvement can't be predicted, it is clear that computers will continue to become more powerful for many years to come.

Ease of Use.

Computers have become easier to use not with steady predictable improvements, but in periodic leaps. Design of the interactive terminal and later of gui interfaces mark two of the major leaps in user orientation. Interfaces get better; analogies improve. As designers get more experience, they learn what features people have difficulty with. New interfaces address those specific aspects.

Power-to-Cost Ratio.

With most commodities, such increasing power would mean a corresponding increase in price. Not so with computers. Computers will get faster and easier to use while they get cheaper. What used to cost millions of dollars now costs only a thousand or two. Over the past decade the price of a new personal computer has remained relatively stable not for the same machine, but for the vastly improved machines available each year. Perhaps the price has even fallen slightly. Certainly, it has fallen if the price is adjusted for inflation.

Ubiquity.

Whatever your field or interest, computers will play an increasing role in your work. New tools are developed every year tools that will soon seem indispensable. The tools will come in increasing varieties; your co-workers will use them; your competitors will use them; and *you* will have to use them to keep pace.

Software Evolution.

The software you use now (whatever it is) will soon be obsolete. It will be replaced by new versions, competing products, and even totally new tools.

26.1.2

Users Must Adapt

Hardware and software compatibility will force users to follow the trends. It will not be possible for users to say, "I'm happy with what I have. I'll just use this old system." What will cause this unreasonable-sounding restriction? Consider the following interlocking trends:

hardware is improving

software producers take advantage of those improvements.

For any product, the designers are eventually faced with a dilemma: Software designers want their products to be as universal as possible. But they also want to add new features that depend on advances in hardware or even other software. For example, they may want to add a very time-intensive feature, or convert a formerly black and white product to color, or use a feature built into a new operating system. The designers try to write software that is as flexible as possible. For example, many applications check to see if the machine has a color or black and white monitor and then display the appropriate images. Eventually the designers simply bite the bullet and admit that a new version of their software

requires a machine made recently. One common problem is memory size: new software products often take advantage of recent advances in memory size. But this means that the software will not run on old hardware. Hardware designers face a similar challenge. Each new hardware design may require compatible software. At the very minimum, the new hardware may not be useful until new software specifically uses its new features. Eventually the new hardware designers say: to use our machine you must have a software version at least as new as x , y , and z .

The user who chooses to stay with either one computer or one version of an application is eventually constrained to use only older versions of the other. Ultimately, the user will find this unsatisfactory. Both software and hardware suppliers will eventually stop supporting a product. Just as it is difficult to get replacement parts for a 1934 Ford, it is difficult to get new parts or support for old hardware and softwarebut the need happens within a much shorter time span in the computing world. Eventually your computer will die of old age. You will need to replace itand your software as well. In all probability you will need to do so

even sooner, either to maintain compatibility with co-workers or others with whom you wish to share data and information, or to take advantage of a new tool.

26.2

Learning to Use New Software Tools

They know enough who know how to learn.

HENRY ADAMS (*The education of Henry Adams* (1907))

Unfortunately the software you have used for this course represents only a small fraction of the total available. Even more unfortunately, the particular software you have been using will certainly be obsolete within a very few years. That is not a criticism of the software; it is a statement about the nature of the world of computing. The same would have been true no matter what software you had learned. Fortunately, you have not spent as much effort on the software as you have on the general problem-solving methodologies. You should now be prepared to move to another environment with a minimum of problems.

It is inevitable that you will not only continue to use computers, but will need to use new applications and new versions of existing applications. How will you ever learn to use all of these tools? For any application xyz, you could read a book or take a course called “Using Application xyz.” Such books and courses almost always exist. Unfortunately, you will need to use many new applications during your career. You will soon get very tired of these courses. In fact, far too many such courses and books already exist. Fortunately there is another approach. This uses a basic rule that applies generally to these new tools:

You already know how to use them!

This is the approach you have used throughout this course. Any two applications probably have more common characteristics than they have distinguishing ones.

*When you encounter new applications
Focus on the similarities with older ones.*

This advice applies to all new circumstances: new versions of existing software, similar applications on other machines, and totally new tools. The tool you know will always serve as a model or analogy for the tool that you do not know.

Every machine has essentially the same capabilities. Within any group of software tools, each application has a common core set of capabilities. As you move to new environments, look for the same capabilities. The names may be different; the interfaces may be different; the speeds may be different; but the basic set of operations should be very similar. When you are confronted with a new piece of software, start with the assumption that it is very similar to one you have already used. Start with a small document or task and seek out the parts of the system that are the same. Remember that most (hopefully) applications include the important software tools:

Help

Undo

Save (and backup)

Laboratory Unit 19 and *Laboratory Unit 14* ask you to use this approach to investigate two new environments. The first explores the use of a very similar environment. For most students, this will mean either the *Windows* or the *Mac OS* environments whichever they have not used to this point.

Not all “new” environments represent the future. On the contrary, some represent the past. The second “futures” lab investigates a “new” kind of environment, the *command-line* environment. The approach taken within this text has assumed a gui interface throughout. Command-line interfaces are from a much older era one in which computer graphics could not provide a time-practical interface for computing: They were just too expensive. Command-line interfaces do not use graphical analogies such as grabbing an object or dragging it to the trash. Instead, the user provides written instructions or commands telling the computer what to do. These commands strongly resemble the commands used in programming in higher-level languages or the algorithmic commands introduced in Chapters 24 and 25. They provide essentially the same set of capabilities as the gui commands. The only serious difference from the perspective of the new user is the method by which the user gives the command: writing the command rather than manipulating graphical objects. However new users do tend to find these more difficult for two reasons:

The user must remember the command rather than select it from a menu.

Spelling and “typo” errors require redoing the commands.

Once again the user should simply remember that the commands

are essentially the same. Ask yourself

What would I call the action I am requesting?

and most important:

Do not forget the help command.

26.2.1

On Reading a Manual

Programming necessarily requires a language. The manual describing the application language is an essential tool for effective programming in that application. Every application has a manual that explains each instruction: its syntax, its semantics, how it is coded, the restrictions on its operands, and so on. Most popular applications have several alternative manuals produced by independent authors. Learning to read the reference manual is an essential skill.

Often a quick look into the manual creates the sensation that reading the contents must be a very formidable task, but it really is not difficult if you approach it properly. It is not necessary to read the entire manual. Remember that

a reference manual is just thata reference manual. It is not a textbook. It contains very little, if any, “how-to” information. The manual simply provides answers to questions that the user needs to ask. The key to using the manual effectively is having a well-formed question to ask. “I don’t know where to start” is not a well-formed question. Use a reference manual to answer specific questions about the exact syntax, the specific results, the legal operands, and so on, of a specific instruction. First develop a general generic algorithm, perhaps in pseudocode, and then convert the algorithm into the language of the application. Then, and only then, look up and fill in the details of specific commands.

The first unfortunate aspect of most manuals is that they are usually organized by command name. That organization is fine if you know the command you need, but terrible if you do not. For example, if `substring` is not called by that name in your specific application, you will not find it in the index under “s”.

Finding Command Descriptions.

You are not lost in new environments. Use the problem-solving approaches you have learned to find the command you want. The techniques apply to both online and hard copy manuals.

1. First, use the index to look up your best guess of the command name.
2. Look in the table of contents for general topic areas that might contain the command you need.
3. Distinguish between deictic commands which probably involve a mouse and general operations which use the menus.
4. Peruse the manual frequently, simply looking at command names or other details that attract your attention. Eventually you will develop a general sense of the names of available commands.
5. Peruse the menus. Look for commands that seem relevant to the

question.

6. Look these up in the manual.

7. Be sure to watch out for variants of commands such as “+” and sum.

26.3

Other Types of Common User-Oriented Software

The software tools you have used while solving the problems posed in the previous chapters represent some of the most common end-user applications; but they represent only a small fraction of the total software available. The following sections list a few of the others. In each case, remember: learning to use the new tool is just an extension of using tools with which you are already familiar.

26.3.1

Personal Finance Programs

Easy-to-use computer programs can take care of writing your checks and balancing your checkbook at the end of the month. They can automatically generate checks that you write on a regular basis; keep track of your investments; make mortgage or auto loan payments; catalog tax-deductible expenses; generate budget reports; and much more.

26.3.2

Tax Calculations

Many students at this point have not had the joy of paying taxes, or perhaps they have only used the short form. But they have certainly all heard the complaints of others as April 15 approaches each year. Tax programs can take care of much of the drudgery. Although the taxpayer still must find and record all of the receipts, the program takes care of all the arithmetic, copying results from one column to another, recording the tax payer's ID on all pages after the first, comparing alternate methods, and allowing you to undo mistakes. Some tax programs can even work in conjunction with the personal finance programs, using the personal finance data summaries as input for tax calculations.

26.3.3

Networking Software

Software exists that enables you to connect to other computers on the Internet and over telephone lines. You can turn your computer into a fax machine. You can sell stocks and buy tickets to the symphony through commercial computer services such as *CompuServe* or *America Online*. Some software can even work behind the scenes. For example, you can ask a program to fetch information or back up your disk in the middle of the night while you sleep. Others work in direct conjunction with other software: some financial packages will pay your bills electronically, eliminating the need for checks.

26.3.4

Office Management

Calendar programs keep track of your appointments. They can beep at you to remind you that it is time to head to the boss's office or list all the appointments you must cancel next week when you take a vacation. Electronic phone books not only keep the latest phone

numbers handy, but dial the phone for you.

26.3.5

Databases and Information Archives

Several encyclopedias, almanacs, and other collections of information now reside on disk accessible to your computer. With these tools, the user has entire libraries of information right on the computer desktop. Computerized books include everything from cooking instructions to anatomy lessons. Many services provide daily (or even more frequent) updates of information with time value (e.g., stock prices).

26.3.6

High-End Graphics

The graphics tools referenced in this text only scratch the surface of what is available. Animation tools can create realistic cartoon image sequences; engineering tools generate three-dimensional views from any angle using a single representation, and *morph* programs transform one object smoothly into another right before your eyes (so much for “seeing is believing”).

26.3.7

Sound

MIDI (Musical Instrument Digital Interface) programs allow computers to accept input from piano-like keyboards (in theory any musical instrument is possible) and produce output sounding like any musical instrument. Voice recognition programs allow users to interact with computers without touching them. Human voices control the computer actions (you probably have used such programs when calling a company for information about a product or questioning your bill). Digital speech output has reached very high levels, often sounding like real human voices.

26.3.8

Specialized Personal Software

The list goes on and on. A rough rule of thumb is: “if you can imagine it, someone has made it.” Software is available for any purpose you can think of. You can get software to help you select a name for your baby, write your will, or create fancy invitations to parties. And if it doesn’t exist today, it probably will exist tomorrow.

26.3.9

Specialized Career-Oriented Software

All of the software described in this text is general purpose, with applications in every field or career. Specialized software now exists for virtually every career or academic field. No matter what your interest, the appropriate specialized software can save you time and effort.

Many academic departments include the use of these products as a routine part of their courses. Thus a physics course may use a tool for measuring and comparing electronic signals from a testing device. A sociology course may use statistical software, and an art course may employ advanced graphics tools.

26.3.10

The Bottom Line

No matter what your career, no matter what your avocation, you will be using computer tools in the future. And you will be using an increasing number and variety. Fortunately, you know how to use them: you have the basic knowledge needed to explore these new tools as you encounter them.

26.4

What Other Courses Should I Take?

For some students, this first computer science course will also be their last. For many others, it is only the beginning. What you will need depends on your personal life and career goals. The most important aspect is whether you want to pursue a career directly related to computer science or if you will simply be using computers in your job.

26.4.1

Computer Science and Computing

Much of the distinction between computers and computer science can be described by paraphrasing the user-designer distinction used throughout this text. Computer scientists tend to be the designers. Computer users tend to be, as their name suggests, users.

Computers are appropriate tools for both groups. As stated earlier, computer science is the formal study of algorithms and of algorithmic problem solving. In this course, you have used many of the methods of computer scientists, but you have done so in the context normally associated with more casual computing. Students who want to learn more about:

algorithms

solving complicated problems

or the design and creation of software and hardware

may want to investigate computer science in more depth. Students who see themselves as users of computers in support of another fundamental interest will probably fit the “users” paradigm. While both groups may benefit from additional computer science or computing courses, the directions they take will vary.

Computer Science Students.

Many schools offer both bachelors and advanced degrees in computer science. Many also offer computer science minors or concentrations. The traditional course of study at the bachelors degree level starts by learning to use a higher-level language such as *Pascal* or *C++*. In terms of the topics covered in this text, these languages are most like the languages discussed in Chapters 24 and 25. After dealing with user-oriented applications, many students see these languages as low level: it takes a lot of code to achieve a result. But this low level also brings power: The

programmer of a higher-level language is not restricted to cells in a spreadsheet or drawing tools or any other subtask. Most user applications are designed to make specific tasks easy. The side effect of this is that they make other tasks more difficult. Higher-level languages provide programmers with universal flexibility at a price of a lower starting level.

Armed with this flexibility, computer science students can go on to study other aspects of computing:

Data structures: How can you extend the basic linear, hierarchical, and matrix structures in useful ways? Most students rapidly recognize many structures as essential tools needed to make usable computing systems.

Analysis of algorithms: Can you determine in advance how long a program will run? Is there a better algorithm? Many students are surprised to learn that they can select the better of two algorithms without even writing the program.

Theory of computation: Are there functions that cannot be computed? Surprisingly, not only is the answer “yes” but you can prove that specific functions cannot be computed.

Organization and architecture: How is a computer actually constructed? It turns out that every data item is ultimately represented as binary values, 1 and 0, and every operation can be described in terms of Boolean logicliterally every operation including those which appear to have nothing to do with logic, such as arithmetic or simply saving information in memory.

Operating systems: How do you build the gui interface? What services should you provide a user? Have you ever noticed that in some systems you can ask the computer to print a fileand then return to your other tasks before the file is printed? How did it do that?

Specialty areas: Certain applications are of such broad interest that computer science departments offer courses in the theory that underlies their construction: computer graphics, artificial intelligence (e.g., understanding natural language or playing chess), database design, compilers, and others.

For Non-CS Majors.

Even students who do not wish to explore the field of computer science much further may find other computer science courses useful. The preface describes many of the courses commonly aimed specifically at nonmajors. Interested students may want to reread the appropriate sections now as a guide to potential “next courses.” In addition, some departments offer specialized courses that cover specific software tools.

26.4.2

Other Fields and Computing

Some academic departments offer their own courses involving computing tools. Usually such courses include the use of software tools in support of a more general concept. Occasionally departments offer a course dedicated to computer tools. Check your catalog for descriptions of learning experiences within your own

department.

26.5

Will Computers be Used for Good or Evil?

This text has included frequent examples of ethical debates within the computing community. These debates focus on human behavior, not on computers. Some individuals want to ask questions they see as more fundamental: At the bottom line, are computers good or bad? Will they serve more good purposes or bad? Have they brought humankind happiness or misery? Few human inventions are inherently good or evil. Virtually every invention has both good and evil applications. The internal combustion engine enables humans to travel to places they could not otherwise visit, enables life-saving supplies to be delivered to those in need, and facilitates shipment of commodities so that we can eat fresh vegetables all year round. On the other hand it is a major source of pollution and can be used as an instrument of destruction (e.g., a bomber).

Consider the following story:

*An inventor in ancient China brought his new invention to the emperor. He said his invention would enable men to fly through the sky, to travel great distances in a short time, to get to the other side of mountains easily, and a myriad other wonderful uses. Unimpressed, the emperor ordered the man executed. Why? Because the invention would enable enemies to get over the Great Wall and invade China.*¹

This text contains several examples of similar dilemmas. In each case it is not the computer or the computer program that must be evaluated, but the use to which humans put those tools. The first rule of computing ethics (see Section 2.4.2) suggests that questions about the good or evil in computers are misdirected. The computer revolution does create two unusual ethical issues, however:

More Harm Faster.

Many issues seem to follow from the simple fact that computers enable humans to have a greater impact in a shorter time. Database tools enable users to snoop into other persons' business far deeper, broader and faster than by conventional means. On the surface the difference is only one of magnitude. But prior to the invention of computers some privacy issues just didn't seem important. It just was not practical for the Internal Revenue Service to search large numbers of databases for your social security number. Few people considered privacy to be in danger. As computers enable more people to do more things faster, these new issues will seem important.

Abuse of Intellectual Property.

Computing has brought the limitations of our intellectual property rights laws to the forefront. Without a computer, it would cost almost as much to copy a book on a duplicating machine as it would to buy the book. Before digital processing, a pirated tape recording was never of the same quality as the original. Computing

systems make it possible to create exact copies of intellectual property at little or no cost. Clearly our laws need to evolve rapidly to keep up with changing technology.

Exercises

26.1 Find a catalog from a computer mail order company. For each set of criteria below, find descriptions of two software packages that match the criteria:

- (a) might be useful in your major,
- (b) might be useful to you at home,
- (c) don't seem to match any software described in this text,
- (d) sound completely ridiculous.

(table continued on next page)

1. Adapted from Ray Bradbury's short story, "The Flying Machine," contained in *The Golden Apples of the Sun* (New York: Doubleday, 1953).

(table continued from previous page)

- 26.2 Look at the list of computer science courses at your institution. List the three courses that look like they might be of most use to you. List the least likely one.
- 26.3 Find a course within your major that makes heavy use of computers.
- 26.4 Write an essay on the potential of computers being used for enhancing evil.

26.6

Summary

Whether this point marks the beginning or end of your formal training in computers and computer science, you now possess the essential basic tools you will need for learning about new tools and environments. You can use this knowledge as a starting point to begin a systematic study of the subdisciplines that make up the field of computer science or as a basis for your future learning in the field. The only item that is truly certain is that you will need to continue to grow and expand your knowledge.

Important Ideas

command-line interface

Index

A

abacus 12

Abbot and Costello 32

abbreviation 328-333, £

See also: macro.

absolute address 282-284

abstraction 6, 8, 33, 327-337, 340-341, 442-448

and new applications 458-459

and scope 267

procedural 444-448

access privileges 57

account £

accuracy 145

acronym 7, 328, 329

ad hoc 76

Adams, Henry 458

addend 151

address 1, £

absolute 282-284, 338, 415, £

Internet 387

relative 282-284, 415, £

www £
addressing
 in spreadsheet 283
algebra 210, 315
Alger, Horatio £
algorithm 4, 8, 183-199, 201-210, 419-434, £
 definition of 183
 development 261
 in a spreadsheet 203
 incorrect 207, 236
 representation of 422
algorithm development 217, 224
alias 63, 332, 332-333, £
 to a file £
 to user £
aliasing 114
align £
al-Khowârizmî 184
ALT (key) 330
alternate mode key 86
alternative 289
ambiguity 6
America Online 393, 398, 461, £
American Standard Code for Information Interchange £

amortization 52, £
anachronism 45, 159
analog 12
analogy 6, 8, 33, 66, 112, 139, £
analysis 5, 8
analysis of algorithms 463
Anarchie 391, 399, £
anchor 405, £
And (Boolean operator) 308
See also: conjunction.
animation £
Annenberg, Walter 346
anti-bugging 238
appearance 231
AppleMail 333
application 11, £
See also: specific types (spreadsheet word processor, etc.)
career-oriented 462
database 461
graphics 461
network 391
networking 461
office management 461
personal 462

personal finance 460
sound 462
specialized 391
tax 461

application-independent approach £
approximation 6, 8, 168, 243
Archie 391, 392, 399, £

archival storage 44
argument 155
arithmetic series ix

Arrange menu £
array 254
arrow key £
artificial intelligence 43, 87, 464
ascenders 86

ASCII. *See:* American Standard Code for Information Interchange

Asimov, Isaac 35

assignment 157, 158
assignment statement 424

ATM. *See:* automated teller machine.

atomic 196
attach 61, £

automated teller machine 393

average 447

average (built-in function) 161, 447

B

background £

backspace 16

backup 23, 28, 459, £

Bacon, Francis 204

bandwidth 385

Banner £

bar chart composite 354

Barber of Seville 242

Barthes, Roland £

b-board. *See:* bulletin board.

Beecher, Henry Ward 437

Bierce, Ambrose 129, 327

big picture £

BigByte Computer Company 260

binary digit. *See:* bit.

binary function 162

binary search 325

BinHex £

bit 72

blank (character) 81

BMP 126

boiler-plate 339

boing (sound) £
bold (style) 77
bookmark 379, £
Boole, George 307
Boolean algebra 314, 315
bootstrap £
border £
bottom-up construction 227
Bradbury, Ray 465
branch 94, 366
brother (node) 96
Browning, Elizabeth Barrett 253
browse 371
browser £
bug 237
build £
bulletin board 388
button 372, 402, £
 label 412
Button Info £
byte 44

Note: £ designates items also referenced in the lab manual.

C

C 217, £

C++ vii, 463

cache 44

calculate 149

calculation multi-valued 175

calculator £

calculus 352, 363

call 448

Camus, Albert £

cancel 27

Canvas (drawing program) 118

card 373

card field £

Carnegie, Andrew 31

carriage return (character) 82

Carroll, Lewis xiv, 32, 83

Cartesian coordinate 352, 430, £

cat 197

cd (Unix command: change directory) £

cell 157, £

Cervantes, Miguel de £

change 154

character £

 invisible 81

Character Map £

chart 343, £

 and mathematical relations 361

bar 346-347, £

line 350, 354, £

multiple line 353

pie 348, £

politician's 359

X-Y line 356

X-Y scatter 357

check box £

check point 244

Chicago (font) 84

child (node) 96, 366

choose 430

Chooser £2

chose £

Church, Alonzo 4, 186

Churchill, Winston 240

Church-Turing conjecture 4

Cicero 455

circular reference 208, 210

circumflex 160, £

Clancy, Tom 148

ClarisWorks 15, 116, 118, 136, 157, £

Classic Mac £

Clemens, Samuel 148

click £

 double 329

ClickArt 53

client 56, 389, £

clip art 125, £

clock £

close 19

close-box £

closed lab £

Cobol 329

coding 226

 and spreadsheets 228

collapse £

color 113, £

column 158, £

command 422

 finding 460

keystroke £

See also: statement.

COMMAND (key) 330

command box £

command line 459

command mode £

comment 232

comment statement 442

Common business oriented language. *See:* Cobol communication
privacy of 69

Compact Pro £

comparison

- multiple 353
- of the whole 348
- simple 345
- with several values 347

compilation 246

compile 402-403

compounding 278

CompuServe 393, 398, 461, £

computability 149-151

computable 152

compute 149

Computer applications (course) vi

computer architecture 464

Computer literacy (course) vi

computer organization 464

Computer-ADE 8

computing

 and the future 455-466

concat 197

See also: concatenation.

concatenation 197, 427

conditional £

 anatomy of 290

 and hierarchies 324

 in higher level language 440-442

 in spreadsheet 292, £

 multiple 295

 nested 304

conditional action 290, 291, 292

conditional statement 292

conditional step 289

conditional test 291, 292

Confucius v, £

conjunction 308-309

 in spreadsheet 309

construction

 incremental 433

container 424, 426, £

continuum 113
CONTROL (key) 330
control character £
control key 330, 410
control panel £
control structure 257, 437-453, £
copy 40, 63
Corner, D. xv
corporate ladder 98
Courier (font) 84
course
 computer science 463-464
C-prompt £
cracker 397
create 40, 92
cross reference 375
Cupper, R.D. 15
cursor 17, £
cut 21
cut and paste 21, 103
D
daemon 385
Darrow, Clarence 343
data 17, 18

derived 179
incorrect 237
initial 179
irregularly distributed 355
shared 56-59, 390

data collection 144

data entry 142, £
data entry mode £
data reduction 344, 345

data retrieval 141

data structure 463
hierarchical £

database 11, 129, 136-145, 367, £
limitations of 368
logical expression 321
online 390

database (application) 137, 143-144, 165

data-recovery 30

date (built-in function) 263, £

daughter (node) 96

dBase (Data base application) 136

DDEExpand £

De Morgan, Augustus 316

De Morgan's law 316

de Saint-Exupéry, Antoine £

debugging 202, 203, 204, 230, 235, 237-239, 250, 267, 323, 433

deictic 329

delete 16, 40, 139, £

DELETE key 27

delimiter 267, 405, 423, 438, 442

See also: scope.

Denning, P. xv

dependent value 155

dependent variable 155, 352, 354

derivative 352

Descartes, René 55, 352

descendent 96

descender 86

design 5, 6, 8, 141, 142, 406, £

top-down £

design methodology 143-144

design mode £

designer 50, 141

DeskDraw (drawing program) 118

DeskPaint (painting program) 116

desktop 47, 49, £

development 6, 8

Dewey Decimal System 33

diacritical mark £
diagnostic 7, 298
diagnostics 8
dialog 27, £
dialog box 27, £
dimensionality 113
Dingbat (font) 87
dingbat (printer's symbol) 126
dir (command) £
direction 6, 8
directory 46, 61, £
disjoint covering 320
disjunction 309-311
 inclusive 310
disk 47
 floppy 44
 non-removable 44
 personal £
 shared £
disk drive 44
 external 44
 internal 44
display 180
Disraeli, Benjamin £

distribution list 388, £
distributive laws 316
divide and conquer 1, 6, 8, 218, 381, £
division
 symbol for 160
do 258
document 12, 14, 39-53
 public 389
documentation 180, 228-230, 268, 340-341, 433-434, 442, 452
designer 229
guidelines 230
procedures 452
user 228
domain 10, 156, 387
doMenu (Hypercard command) £
Dorner, Steve 333, £
DOS 339
double negation 316
double-click £
Dow Jones 345, 384
down £
download 389
drag 430, £
draw £

draw tools 115, 118-119

drawing £

drive 45

due vs. owe £

Dungeons and Dragons 372, 414

Duplicate (command) £

dynamic £

dynamic link 371

E

edge 94, 365

edit £

EDSAC. *See:* Electronic Delay Storage Automatic Calculator.

efficiency 250

Einstein, Albert £

Eisenhower, Dwight D. 201

Electronic Delay Storage Automatic Calculator 235

electronic mail xi, 60, 60-66, £

element 130, 133

Eliot, George 271

ellipsis 207

else 440

Email 386, £

Internet 63

See also: electronic mail.

empirical method 7, 8, 190

empirical result 204

empirical verification 5

empty 83

end 438

end of file £

English measure 169

equation

 in spreadsheet 158

erase £

ERIC 391

error 154, 155, 235-251

 assumption of 240

 cascaded 248

 catastrophic 42

 computer 154

 fence post 266

 file 249

 logical 249

 machine 248

 offsetting 239

 period £

 removal 237

 run-time 247, £

sources of 235
syntax 246, £
typographical 154
zero divide 298

estimation 168-175

ethics and computing 35-37, 50-53, 58-59, 68-69, 88-89, 127, 144-145, 395-397, 464-465
breaking and entering 397
data collection 144
first rule of 35
harassment 68
intellectual property 51
misleading charts 359-360
plagiarism 88
pornography 127
privacy 68, 347
viruses 58

Eudora (mail system) 333, £

evaluation 7, 8

event 422

evolution
of software 457

Excel 157, £

exclusive or 310

excuse 30

exemption

 personal £

exit 409

expand £

experience 7

experiment 31, 204

experimentation 7, 8

exponential 361

exponentiation

 symbol for 160

expression 162

 ambiguous 188

 and algorithm 191

 arithmetic 189, £

 arithmetic and trees 196

 Boolean 380

 nonarithmetic *See:* expression, nonnumeric.

 nonnumeric 196, 263

 parenthesized 191

external memory 43

F

facsimile machine 126

factorial 275, 439

false 291

FAQ. *See:* frequently asked question file.

fax. *See:* facsimile machine.

Fetch 390, 398, £

Fibonacci series 280

Fibonacci, Leonardo 280

field 133, 134, 428, £

file £

file conversion £

file drawer £

File Manager £

File menu 41

file server xi, 56-58, 390, £

file transfer protocol 390, £

FileMaker Pro (Data base application) 136

files 46

fill (graphic command) £

fill characters 248

Fill down 258, £

Fill right 258, £

Find 40, £

Finder £

finding documents 41

finger 392, £

finite 189

Fitzgerald, F. Scott 71

flat data base £

flexibility 11

floppy disk £

flowchart 256

folder 46, 48, 61, 98, £

font xiii, 84, £

(*See also* individual font names: Times, Helvetica, etc.)

ascenders 86

descender 86

mono-space 84

proportionally spaced 85

sans serif 84

serif 84

footer 339

For 258

For next 258

Ford, Henry vii

foreground £

form 339

formal lab £

format 144, 178, 180, £

document £

paragraph £

Formula translator. *See:* Fortran.

Fortran 194, 329

forward 61

FoxBase (Database application) 136

FrameMaker 53

Franklin, Benjamin 452

freeware 395-397

Frege, Gottlob 242

frequently asked question file £

Frost, Robert 56

Frye, Northrop £

ftp. *See:* File Transfer Protocol.

function 12, 147-165, 167-181, 354, 448, £

Bolean 313

built-in 161, 333, 447

computable 151

definition of 148

inverse 149

nonnumeric 313

one-to-one 149
recursively defined 274-281
representation in spreadsheet 179

G

Galileo 24
game 372
garbage in; garbage out! 237
Garland, S. 15
gateway (in computer network) 384
Gateway Lab xii, £
gender-specific terminology xlv
General controls (control panel) £
generality 11, 178
generalization 201
Geneva (font) 79, 84
GEnie 393, 398, £
geometric series ix, 278
GeoWorks 15
Get Info £
Gibbs, Willard 39
GIF. *See:* graphics interchange format.
giga- 45
gigabyte 45
GiGo. *See:* "garbage in; garbage out!"

Gillette, King Camp 396
go to £
Gödel, Kurt 3, 242
Gödel's Theorem 240, 242
Goethe, Johann Wolfgang von 217
Goodman, D. 434
Gopher 376, 382, £
grandchild (node) 96
grandparent (node) 96
Grantham, Brad 79
granularity 14, 19, 20, 27, 63, 119, 143, £
graph 109, 343, 365-382, 406-411, £
 limitations of 367
See also: chart.
graphical user interface x, 456, £
graphics 79
graphics interchange format 126
grave £
Great Works (integrated application) 15
grid £
Gries, D. xv
group 20, £
 abstract 438
grouping

as abstraction 335

gui. *See:* graphical user interface.

Gutenberg, Johann 81

H

hacker 397

Haldeman, H.R. 27

Hall, Tom L. 434

halting problem 242

handler 424

harassment, electronic 68

hard code 179

hard disk 44, £

hard-wired 394, £

harpsichord (sound) £

Harvard (outline format) £

Hawthorne, Nathaniel 167

Hazlitt, William 7

header 102

Hello World program £

help 23, 24-27, 245, 371, 375, 408, 459, £

application-specific 25

balloon 24, £

context sensitive 24

context-sensitive £

interactive 24, 29
limitations on 26
menu 25
online 24
tutorial 25

help dialog 27
help facility 24
help wanted column, on-line 391
helper program £
Helvetica (Switzerland) xiv
Helvetica (font) xiv, 79
hide £
hierarchical 218
hierarchy £
higher-level language. *See:* language, higher-level.
hill climbing. *See:* search strategy. 379
history x
Hollerith, Herman vi

Holmes, Oliver Wendell £
Holtz, Matthew 434
home 373
home card £
home page 373, 378, £
homework £
host 389, £
Hot Java 377, 419, 423
hotlist £
housemate (problem) £
Href (Html marker) 405
href. *See:* hyper reference.
html. *See:* hypertext markup language.
http. *See:* hypertext protocol.
human factors 35, 456, 457, £
Hustedde, S. 434
Huxley, Thomas £
hyper (prefix) 370, 415
Hyperarchive 380, £
HyperCard 116, 377, 403, 419, 423, £
hypercube 370, 415
hypermedia 109, 365-382, 386, 401-416, £
(definition) 370

application 367, 375
construction £
limitations of 414
navigation 365-382
programming environment 377
hyperreference £
hyperspace 415
hyperstory 409
HyperTalk 423, £
hypertext £
hypertext markup language 404, £
hypertext protocol £
hypothesis 205

I

I/O. *See:* input-output.

IBM. *See:* International Business Machines.

icon £

if 293, 440
if statement £

image 114, £
image viewer 126

ImageWriter £

import 124

include 61

income
 gross adjusted £
 taxable £
indentation 233, £
independent value 155, 352
independent variable 354
index 320, 368-370
 limitations of 368
infinite loop 208
infinite regression 81
infix 162
InfoMac £
information 18
 structure 74
information processor 18
information superhighway 383, 384
InfoSeek 382
input-output 223
insertion point 17, 139, £
integration 286
intellectual property 52
interactive 180, 371
interest £
 simple £

interest rate £

interface x, 332

 command line 459, £

intermediate value 232

internal memory

See: random access memory.

Internal Revenue Service £

International Business Machines x

Internet 127, 375, 383-394, £

 address 387

 providers 393

interpretation 130, 402, 404-405

 incorrect 236

intractable 13

Intractable problems 13

Introduction to Computer Science (course) vii

invariant 244

inverse 149

IRS. *See:* Internal Revenue Service. £

IsNumber (spreadsheet command) 301

IsText (spreadsheet command) 301

italic (style) 77

iteration 218, 253-269, 271-287, £

 and algorithm design 265

and debugging 267
double 264
for creating pictures £
in higher level language 437-440
in spreadsheet 257

iterative 218

Itty Bitty Machine 290, 344

J

Jabberwocky xiv

Java £

Johnson, Lyndon Baines 203

join (concatenation) 197

join (relational operation) 323

joint photographic experts group 126

JPEG. *See:* joint photographic experts group

justification £

K

Kelemen, Charles xv

Key Caps £

keyword 380, 412

Kilmer, Alfred Joyce 91

kilo- 45, 280

Kipling, Rudyard 223

Koblas Currency Converter 391, 398

Koblas, David 398

L

lab

closed £

formal £

informal £

open £

label 157, £

labor

sharing 202, 203, 250

laboratory manual xii

LAN. *See:* local area network.

language 72

command 338, 389

formal 420

higher-level vii, 420

imperative 422

scripting 338

lasso £

Lavater, Johann Kaspar £

layout 142, £

Leacock, Stephen Butler x

leaf 94

learning

lifetime 458

learning curve 141

lecture xi

library card catalog

 online 390

Library of Congress 33

licence 50

life cycle 141-143

limit 208

limit values 244

limits 13

Lincoln, Abraham 23

linear 72

link 371, £

 conventions for 411

 dynamic 401-405, £

 global £

 local £

See also: pointer.

static £

link (*continued*)

timed 413

list 108, 130

ordered 131

recursive 271

unordered. *See:* set. 130

listserver 388

literal 178

living 158

living document 140-143, 158, 371

See also: interactive.

loan £

loan amortization formula £

local £

local area network 56, 384

local information £

location 123

Locke, John 307, £

log on 393

logic 289, 307-326

Boolean 307

logical expression

and database retrieval 321

logoff (command) £

logon (command) £

loop 256, £

infinite 207, 256

Lotus 1-2-3 vi, 157, £

Love, Michael 70

lower £

lp (Unix command) £

ls (Unix command) £

Lycos 380, 382

M

MacBinary £

MacDraw (drawing program) 118

machine tool ix

Macintosh x, 15, £

Macintosh Basics xv, £

Macintosh OS 459, £

MacPaint (painting program) 116

macro 327-339, £

as abstraction 340-341

construction of 337

definition of 335

limitation 338

recording £

MacWeb £

MacWrite 15, £

mail

junk 68

voice 370

mailbox £

mail-merge £

main memory. *See:* random access memory.

mainframe £

Mallory, George 202

man (Unix command: manual) £

manual

reading 459

Map £

Maple 363

Mathematica 363

mathematics xii

mathephobia 151

matrix 157, 254

max 161, 300, £

maximum £

Mead, Margaret xi

measuring system 169

mega- 45, 280

megabyte 44

Melville, Herman 148

memory 42

external 43

internal 29

long-term 42, 44-45

magnetic 43

main 28

short-term 42

memory hierarchy 44

menu £

Apple £

menu, application £

message 422, 444, £

Message box (HyperCard tool) £

meta- (prefix) 32

metric conversion 169

metric system 169

Microsoft £

Microsoft Word vi, 189

Microsoft Works (integrated application) 118, 136, 157

MIDI. *See: Musical Instrument Digital Interface.*

Miller, George 43

min (built-in function) 161, £

misconceptions 52
mnemonic 7, 329, 448
mod (built-in function) 161
modal £
mode £
 user 142
model 111-128, 406, £
 analog 112
modeless £
modeling 5
modem. *See*: modulator-demodulator.
Modula vii, 217
modular computer £
modulator-demodulator 395
Monaco (font) 84
Moore's Law 456
more (command) £
morph 461
Mosaic 376, 382, £
mother (node) 96
mount 58
mouse £
Mouse (control panel) £
mouse button £

mouse pad £
mouseUp £
move 40, £
MS Windows Paintbrush (painting program) 116
MS Works £
Mulder, M. xi
multimedia 377
multiplicand 151
multiplication
 symbol for 159
multiplication table 264
Mumford, Lewis £
Murphy's Law £
Musical Instrument Digital Interface 462
myth 8, 1, 226, 249
N
naming conventions 48
natural
 number 274
natural language 87-88
 (define) 87
navigate £
navigation 420, £
negation 312-313

double 316

negative number
representation of 299

Nehru, Jawaharlal £

net
See also: Internet.

Netfind 392, 399

Netscape 376, £

Netscape Navigator 382, £

net-surfing £

network
wide-area 57

new 19

New field £

news group 388

news server 388

nickname 63, 328, 331-332, £

node 94, 365, 395
parent 374

Not (Boolean operator) 312
See also: negation.

notes £

null 83, £

range 83

number

natural 150

numeric (outline format) £

Nutthall, Chris £

O

object 14, 39-53, 114, 422, £

and draw tools 118

hidden 120

selection 120

object oriented 422, £

object oriented programming 422

obsolescence viii

Olds, Ransom vii

online 29

OOP. *See:* object oriented programming.

open 19, 40, £

operand 14, 162, £

operating system 41, 464

operation 14

bit 120

relation 138

specification 120

operation, combined 21

operator 14

nonarithmetic 301

relational 345

option key 86

Or (Boolean operator) 309

See also: disjunction.

Oracle (Database application) 136

order 187, 209

organization 47, 232

ad hoc 76-79

hierarchical 91-109, £

logical 45

orphan 410

OS/2 15

outline 100-105, 407, £

Harvard £

representation of 103

outliner £

outliner (application) 104

Output 233

owe vs. due £

P

page 373

page set up £

pagination 78
paint tools 116-117, £
palette 115, £
parameter 338, 446, £
 actual 446
 formal 446
parent (node) 96, 366
parentheses
 nested 192
Pascal vii, 217, 463
passing 446
passwd (Unix command) £
password 58, 389, £
patch 248
patent 51
pattern £
Pavarotti, Luciano 16, 31
payment £
Peano, Giuseppe 274
Peano's Axioms 274
pedigree 94
personal identity number 393
Pete, Laurence J. 71
Phaedrus 419

Phillips, Wendell £

phone £

pi 175, 275

Picasso, Mrs. Pablo 289

Picasso, Pablo £

PICT 126

picture 111-128, 412, 414

algorithmic £

pixel 114, 430

and paint tools 116

operation 120

pixel operations 117

plagiarism 88

platton 17

Play £

plot. *See:* Chart.

point 78, £

pointer v, 1, 46, 63, 102, 109, 125, 332, 371, 372, 376, 386, 401-405, 407-409, 426, £

backward 373, 379, 408, 411

creating 409-411

cross reference 375

dynamic 371, 402

forward 373, 379, 407, 411

generalized 375
See also: link.
table of 368-370, 378
pornography 127
Post Office £
postcardware 396
post-lab £
power-level 11
power-to-cost ratio 456
practice xi, 65
precedence 190, 207
precision 202, 250
predecessor 151
prediction 6
Preferences £
preferences £
prefix 162
preparation
 pre-lab £
pretty £
previous (hypermedia command) 379
Price-Jones, Rhys xv
principal £
print 19

Print manager £

privacy 68, 145

problem £

can't find icon £

cannot read disk £

commands

gray £

commands have no impact £

cut and copy £

data organization 91

disappearance

files £

text £

window £

disk £

phantom £

disk full £

insert disk £

intractable 14

machine is frozen £

machine wants disk it has £

menus £

must close window £

no mail £

no printer output £
paste £
printer cannot be found £
printing £
specification 4
text
 appearance £
 insertion £
text entry £
Two files, one name £
unable to read £
updating 92
what-if 261
window £
 opening £
 writing as 10
problem (definition of) 2
problem analysis 6
problem definition (programming step) 222
problem solving 1, 201
problem-solving technique
 comparison shopping 262
 how many times 262
procedural abstraction 227

procedure 444-448, £

built-in 333

communication from 447

procedure (continued)

communications to 445

size of 452

Prodigy 393, 399, £

product 151

program development

debugging 237-239

testing 240-245

verification 240-245

programming ix, 217-234, 235, 431-434, 452

technique 265

programming process 217

algorithm development 217

coding 226

debugging 230

problem definition 222

testing 230

verification 230

project (relational operation) 322

prompt £

property

intellectual 51

protection, access 58

pseudocode 225, 226, 420

pull down menu £

put £

Put away £

Puzzle £

Pythagorean theorem 193

Q

quadratic formula 193

query 138

question

multi-valued 317

representation of 291

questions 223

problem-solving tools 66-67

quit £

R

raise £

RAM. *See:* random access memory

Rand, Ayn 148

random access memory 29, 42

range 156

read 43

readability 232

read-only 57, 339

Reagan, Ronald £
receive 61
record 133, £
recursion 81, 106, 274-281, £
 and pictures 450
 and what if problems 284
 definition of 275
 in higher-level language 449-451
reduced instruction set computer 329
region 120
register 44
 check 280
regular £
relation 129-146, 167-181, £
 arithmetic 295
 manipulation of 322
relational data base £
relative address 282-284
remote access 389
rename 40
repeat 438, £
reply 63, £
representation 71-89, 178
 ambiguous 190

incorrect 236
responder 424
result
 intermediate 190
résumé £
retrieval 136
retrieval mode £
return (from procedure) 447, 448
return (key) £
reusability £
reusable tools £
reuse 141, 155, 194, 202, 250, £
revert 27
revision 205
Riemann sum 286
Riemann, Georg Friedrich Bernhard 286
risc. *See:* reduced instruction set computer.
Rodin 32
root 94, 408
rotate £
router 57
row 158
rule of thumb 167, 174, 206
ruler £

Russell, Bertrand 242

Russell's Paradox 242

S

sans serif 84

Santayana, George £

save 19, 23, 28-30, 459

scanner 126

Scheme 7

Schneider, G.M. 15

science 12

Science Applications International 372

scientific method 7, 31-32, 204, 237

scope 162, 192, 267, 423, 452

 and parentheses 193

 delimiters 443

 of Boolean expressions 323

 of control structure 438

 of control structures 267

 of object 429

 of operators 192

 of procedure 339, 445

See also: delimiter.

Scott, Ray 222

Scott, Walter 365

scrapbook £

script £

search 40, 390

binary 107, 132

sequential 132

strategy

table lookup 378

tree 107-108

search engine 380-381, 390, 392, £

search strategy 377-380

hill climbing 379

starting points 378

superimposed structure 378

testing 5

select 139, £

multiple objects £

select (relational operation) 322

select region £

selection 17, 81, 139

null £

selection tool £

selector 402

self reference 11, 207, 427

semantics 87

send 61, £
sequence 187, 271
 simple 271
sequential 72
series
 nonnumeric 273
 of sums 353
serif 84
server £
set 130, £
Settings £
shareware 395-397
Shaw, George Bernard 343
Shell, B. 434
show £
Show menu £
sibling (node) 96
signature £
significant figures 168
Singh, Bob £
sister (node) 96
size 123, £
slide-bar £
slope 350

snail mail 60

snap £

social implications 35

software 11, 460-462

ownership 50

See also: application.

solution 188

(definition) 3

evaluation 3

recognition 3

specification 4

sort 138, £

Sound £

Sound (control panel) £

space (character) 81, £

spacing 84

line £

SparcStation 15

special-effect (style) 77

spell check £

spreadsheet 11, 137, 157-165, 170, 257, £

sqrt 194

square 118

square root 194

stack £

statement

assignment 424, 425-429

static £

stationery 339, £

step 187

STIS (database) 391

stored-program computer 50

stream 72

strike-through (style) 77

string 72

character £

empty 83

structure 103, £

ad hoc 408

control 255

embedded 366

graph 109

hierarchical 72, 374-375, 376, 406, 408, 411, £

irregular 73

linear 72, 373, 407

list 108, 130

of manual £

standard 452

table 109

Strunk, W. 89, 100

StuffIt £

style 77, £
(also see individual style names: bold, italic, etc.)

stylus 329

subroutine 335

subscribe 388

subst 197

substring 197

successor 151, 198

Sullivan, Louis Henry 147

sum 151

summation 275

Sun 10, 15

SuperCard 377, £

SuperPaint (painting program) 116

Survey of Computer Science (course) 7

switch

on-off £

Symbol (font) 87

syntax 87, 420

system 39, 41, £

system level 39

T

tab (character) 82, £
table 109, 133-134, 138, 176, 177, 254, 368, £
table of contents 102
table-lookup 320
tag £
 title £
talk £
tape 43
target £
tax liability £
tax rate £
teamwork 67
telecommunication 383-385
telecomputing 383-397
telephone
 connection £
Telnet 383, £
template 339, £
tera- 45
terabyte 45
terminal £
terminal emulator £
terminology
 gender-specific 14

test 205
 conditional 290
test condition 291, £
testing 230, 237, 240-245, 323, 433
 data 242-244
testing plan 241
text
 size of 78
text only £
then 440
theory of computation 463
thesaurus 88, £
Thoreau, Henry David 111
thrashing 109
thread 388
thumb, rule of 284
TIFF 126
tilde £
time £
Time and date (control panel) £
time critical 381
Times (font) xiii, 84
title bar £
Toge, Nobu £

Tolkian, J.R.R. 148

tool 10, 139

abstract 33, 137

brush £

definition 10

reference 78

selection 105, 165

selection of 64, 79, 88, 119-122

visualization 111

tool (command) £

toolbar £

ToolBook 377, 403, 419, 423, £

tools 9-22

and models 115

top-down 218

approach 103

design 224, 406, 433

toy problem 261, £

track ball 329

trash £

Trawl 70

tree 93-100, 106-108, 365, £

anatomy of 93

binary 94

corporate hierarchy 97

decision 99

definition of 106

elimination 96

family 94

See also hierarchical organization.

spanning 367

trend 350, £

trouble shooting £

true 291

Truman, Harry S. 383

truth table 308, 315

try it (as tool) 31

Tubman, Harriet £

Tucker, A. xv

Turabian, K.L. 89

Turing machine 186

Turing, Alan 186

Turner, A. 15

tutorial 371, £

twigs 94

type £

U

umlaut £

unary 312

underline (style) 77